

REASONING ABOUT INFORMATION DISCLOSURE IN RELATIONAL DATABASES

A Dissertation

Presented to the Faculty of the Graduate School
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

Gabriel Mintzer Bender

August 2014

© 2014 Gabriel Mintzer Bender
ALL RIGHTS RESERVED

REASONING ABOUT INFORMATION DISCLOSURE
IN RELATIONAL DATABASES

Gabriel Mintzer Bender, Ph.D.

Cornell University 2014

Companies and organizations collect and use vast troves of sensitive user data whose release must be carefully controlled. In practice, the access policies that govern this data are often fine-grained, complex, poorly documented, and difficult to reason about. These issues make it easy for principals to accidentally request and be granted access to data they never use.

To encourage developers and administrators to use security mechanisms more effectively, we propose a novel security model in which all security decisions are formally explainable. Whether a query is accepted or denied, the system returns a concise yet formal explanation which can allow the issuer to reformulate a rejected query or adjust his/her security credentials. In order to demonstrate the practical applicability of our approach, we implement and evaluate a disclosure control system that handles a wide variety of real SQL queries and can accommodate complex policy constraints.

Our *explainable* security model is based on a new theoretical foundation for reasoning about information disclosure in database systems that we call *disclosure labeling*. Information disclosure is expressed in terms of a set of *security views* that are defined by a human administrator and reveal types of information that are relevant to the security constraints of the system at hand. Disclosure labeling allows us to precisely characterize which subsets of the security views contain enough information to determine a query's answer; such characterizations form the basis for the explanations generated by our system.

BIOGRAPHICAL SKETCH

Gabriel Mintzer Bender was born in Providence, Rhode Island, in July 1987, son of Yvette Mintzer and Michael Bender and brother to Ariel Katz (née Bender). At the age of ten he moved with his family to Princeton, New Jersey, where he remained until he was 18.

In 2005 he set out for college at the University of Chicago, where he learned to brave the frigid winters of Illinois. While at UChicago he became involved in an internship with Pedro Felzenszwalb, who, along with Anne Rogers and Anna Lysyanskaya, was responsible for introducing him to academic research in Computer Science.

Gabriel graduated from UChicago with Bachelor's Degrees in Mathematics and in Computer Science in 2009 and left Illinois for the slightly milder climes of Upstate New York, where he became a Ph.D. student in Computer Science at Cornell University. He joined the Cornell Database Group a year later under the supervision of Johannes Gehrke, who eventually became his advisor. He maintained his residence in Ithaca until the fall of 2013, when he spent a year studying in absentia at the University of Washington.

The rest is a work in progress.

To my family.

ACKNOWLEDGEMENTS

The material presented in this dissertation is the result of a close collaboration with Łucja Kot and my advisor, Johannes Gehrke. I would like to thank both of them for their mentorship, moral support, and intellectual contributions over the past four years.

I'd also like to thank the mentors (both official and unofficial) from all my internships over the past ten years – especially Pedro Felzenszwalb, Anna Lysyanskaya, Xiaokui Xiao, Jessica Staddon, Aleksandra Korolova, Nikhil Singhal and Jon Rascher.

Thanks to my family – especially Yvette, Michael, and Ariel – for their continued support over the years.

Thanks to Magda Balazinska, Dan Suciu, and the University of Washington Database Group for hosting me in Seattle during the last year of my Ph.D.

This dissertation has benefited from the feedback of Dexter Kozen, Rafael Pass, and Johannes Gehrke. The work presented here also benefited from questions asked by Christoph Koch, Ashwin Machanavajjhala, and Úlfar Erlingsson.

The research presented here was supported by Grant 279804 of the European Research Council, the National Science Foundation under Grants IIS-1012593 and IIS-0911036, by the iAd Project funded by the Research Council of Norway, and by a Google Research Award and an NEC Research Award. However, the opinions, findings, and conclusions expressed here do not necessarily reflect the views of the sponsors.

TABLE OF CONTENTS

Biographical Sketch	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vi
List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 Disclosure Labeling	3
1.2 Explainable Security	4
1.3 Layout of the Dissertation	8
2 Related Work	9
2.1 Queries and Views	9
2.2 Answering Queries using Views	10
2.2.1 Conjunctive Queries under Set Semantics	10
2.2.2 Finding Query Rewritings	13
2.2.3 Query Pricing	15
2.3 Database Access Control	16
2.3.1 Non-Truman View Rewriting	19
2.3.2 Truman View Rewriting	20
2.3.3 Inference Control	21
2.4 Security and Privacy Potpourri	23
2.4.1 Programming Language Security	23
2.4.2 Reasoning about Security Policies	24
2.4.3 Usability of Security Permissions	25
3 Disclosure Labeling	27
3.1 Problem Statement	29
3.1.1 Notation and Terminology	30
3.2 Disclosure Labeling	31
3.2.1 Disclosure Orders	32
3.2.2 Disclosure Lattices	34
3.2.3 Disclosure Labelers	39
3.2.4 From Labelers to Security Policies	46
3.3 Generating labelers	48
3.3.1 Downward Generating Sets	49
3.3.2 Generating Sets	54
3.4 Labeling Conjunctive Queries	58
3.4.1 Single-Atom Case	60
3.4.2 Multi-Atom Case	66
3.5 Implementation	69

3.5.1	Representing Disclosure Labels	69
3.5.2	Representing Security Policies	71
3.6	Labeling in Practice	73
3.6.1	Reviewing Facebook’s APIs	74
3.6.2	Experimental evaluation	75
4	Disclosure Labeling on Multiplicity-Sensitive Queries	80
4.1	Core Query Language: Definition and Formal Semantics	82
4.1.1	Datasets and Queries	83
4.1.2	Equivalence, Homomorphisms and Foldings	87
4.1.3	Equality Predicates	91
4.2	Query Rewritings	95
4.2.1	Rewritings and Expansions	96
4.2.2	Finding Rewritings	99
4.3	Disclosure Labeling under Combined Semantics	101
4.3.1	An Order Based on Equivalent Rewritings	102
4.3.2	Labeling Single-Atom Queries under Combined Semantics	104
4.3.3	Query Dissection	106
5	Towards Practical Disclosure Labeling	111
5.1	Filter-Project Queries	113
5.1.1	Rewritings of Conjunctive Queries	116
5.1.2	Measuring Disclosure for Filter-Project Queries	121
5.2	SQL Query Language and Semantics	124
5.2.1	Terms, Formulas, and Queries	124
5.2.2	Query Evaluation	126
5.2.3	Free Variables and Safety	130
5.3	Query Generalization	132
5.3.1	Predicate Generalization	135
5.3.2	Automated Query Rewriting	138
5.3.3	Query Generalization and Disclosure Orders	145
5.4	Condition Graphs	147
5.4.1	Constructing Condition Graphs	148
5.4.2	Correctness for Restricted SQL Queries	150
5.4.3	Correctness for Extended SQL	153
5.5	Summary	156
6	Explainable Security	157
6.1	Policy Formulas and Explanations	157
6.1.1	Generating Data-Derived Formulas	159
6.1.2	Minimal Policy Formulas	165
6.2	Explanations	168
6.3	Query Pricing	172

7 Explainable Security in Practice	175
7.1 Generalization Pipeline	176
7.2 Experimental Evaluation	179
8 Conclusion	187
Bibliography	189

LIST OF TABLES

3.1	Notation summary	32
3.2	Inconsistencies between the FQL and Graph API permissions labeling of User attributes.	73

LIST OF FIGURES

1.1	Database and security views.	5
1.2	Disclosure control system model	6
2.1	Example User relation.	10
2.2	Representation of User table as a set of relational atoms.	11
3.1	Sample dataset, security views, and queries.	28
3.2	Disclosure control system model	28
3.3	Security views and corresponding disclosure lattice.	35
3.4	All relational projections of Contact.	48
3.5	Disclosure labeler performance.	77
3.6	Policy checker performance.	78
4.1	Sample dataset based on the Facebook Apps schema.	81
4.2	Effect of copy variables on query evaluation.	85
4.3	Removing equality predicates through variable unification.	93
5.1	Sample database schema.	112
5.2	Evaluation semantics for SQL terms and formulas.	126
5.3	Truth tables for three-valued Boolean logic.	127
5.4	Sample dataset based on Facebook Apps.	127
5.5	Rules for computing free variables in terms and formulas.	131
5.6	SQL queries handled by our condition graph algorithm.	148
5.7	Condition graphs for queries from Fig. 5.6	149
6.1	A set of security views	158
6.2	Algorithm for computing policy formulas (decomposable case)	162
6.3	Axioms encoding relative information revealed by sets of views.	166
7.1	Computing policy formulas – System Architecture	176
7.2	Query generalization pipeline	177
7.3	Architecture of Prototype System	180
7.4	Performance for ordinary queries	183
7.5	Performance for prepared statements	184
7.6	Performance for complex SQL features	186

CHAPTER 1

INTRODUCTION

Over the past few decades we have seen the emergence of companies like Google and Facebook that collect and store large amounts of personal user information. This data can be used to deliver highly relevant content to end users. It can also be used to match advertisements with users' interests, thereby creating significant value for advertisers.

However, companies risk serious damage to their reputations if they lose control over this data. Fine-grained permissions provide better control over who can access what resources, and as a result, platforms such as Google Android and Facebook Apps have dozens of distinct permissions, each regulating access to a different resource or type of user data. Similarly, commercial solutions such as IBM's *Label-Based Access Control* [9] and Oracle's *Virtual Private Database* [11] give database administrators precise control over which principals can see which parts of a database. Such control is largely motivated by the Principle of Least Privilege [48], which states that principals should be granted the least permissions needed to do their jobs.

Although our emphasis here is on *controlling* what data or resources untrusted programs can access, we should note that the use of permissions (and fine-grained permissions in particular) can be as much about transparency as about control. Apps written for platforms such as Android and Facebook Apps can request any permissions they want. However, users can see what permissions apps want before installing them, and can choose not to install apps that request access to resources or user data that seem excessive or otherwise suspicious. They can even post on the app stores to warn off other users. In these cases, fine-grained permissions force apps to tell users up front which resources they want access to, thereby improving transparency.

Unfortunately, the benefits of fine-grained permissions come with a high cost: fine-grained permission structures are inherently complex and difficult to reason about. To make matters worse, the documentation for these permission structures is often inaccurate or buggy. [15, 28] If a query fails due to insufficient permissions, the path of least resistance is to blindly request more permissions until the problem goes away. As a result, principals frequently violate the Principle of Least Privilege by acquiring permissions that they never use. [28]

The fundamental problem is *not* that current security mechanisms are incapable of encoding complex security policies. After all, there has been a substantial amount of research on database access control, and there are even systems such as IFDB [51] that control both disclosure and information flow. Also, in practice, app ecosystems do have a simple way of controlling disclosure: they allow users to set data permissions. Each permission conceptually regulates access to a specific *view* of the data.

However, existing approaches to disclosure control have some major disadvantages. There is evidence that both users and developers misunderstand and misuse app permissions [28, 29]. As new data is added to the database and old data is put to new and unexpected uses, the permission structures can get out of date and inconsistencies occur. For example, the Facebook permission named `user_likes` confusingly gives apps access to both a user’s “Liked” pages and the languages the user speaks. In short, complex security policies are difficult for humans to reason about. This complexity makes it difficult for developers to determine what permissions their apps *should* request, and makes it easy for developers to request permissions that may look relevant to their apps but are not actually needed.

The problem can be addressed in one of two ways: platform providers such as Google and Facebook can either revert to simpler permission structures or else can build tools

to help principals reason about complex ones more effectively. Although reverting to simpler and coarser-grained permissions is a viable option, doing so gives users less control over who can access their data. For this reason, the development of tools that can help principals use complex permission structures is an important thread of work.

The development of such tools – and the theoretical foundations to support them – is the focus of this dissertation. *Disclosure labeling*, which provides a formal basis for reasoning about information disclosure, is described at a high level in Section 1.1 and is discussed in detail in Chapter 3. On the practical side, *Explainable security* is a new security model built on disclosure labeling that can provide concrete feedback to principals about the permissions that they *should* be requesting; it is summarized in Section 1.2 and is discussed in detail in Chapter 6.

1.1 Disclosure Labeling

On the theoretical side, we propose disclosure labeling as a formal foundation for reasoning about information disclosure. In contrast with most previous models of information disclosure used by the database community, disclosure labeling is *data-derived*. In a data-derived disclosure model, the disclosure associated with any view over the database is a mathematical function of the data needed to compute the view. This is very different from ad-hoc hand-generated disclosure descriptions such as the Facebook `user_likes` permission; a data-derived approach would remove opportunities for human error caused by misleadingly named permissions such as `user_likes`. A further advantage of a data-derived approach is that the information disclosed by a view can be computed *automatically* from the view definition, which reduces the burden on humans and makes the process less error-prone.

However, being data-derived is not enough; for example, a definition of disclosure in terms of the number of bits that are being disclosed (in the spirit of Differential Privacy [26]) is unlikely to be meaningful to a user. Thus we need a notion of disclosure that is not just data-derived, but also *semantically meaningful*, which means that disclosure maps to a concept that the user can intuitively understand.

The third requirement for a disclosure model is that it should be *expressive*. While the security policies for games and social apps are often simple, the corporate world is also becoming increasingly dependent on app ecosystems through BYOD (Bring Your Own Device) solutions [3]. Mobile apps are also used in the military [10]; both of these use cases demand significantly more complex security policies.

1.2 Explainable Security

We use disclosure labeling as the basis for a new security model in which all security policy decisions are *explainable*. Instead of simply rejecting unauthorized queries, our system provides the issuer with concise *explanation* of why the queries were rejected and what additional permissions would need to be granted for a successful execution. The principal can then refine the queries or request additional permissions based on the explanation.

We begin with a generic architecture for access control that is applicable to many modern DBMSs. We clarify how our *explainable* model compares to traditional database security models and identify concrete properties that should be satisfied by explanations for policy decisions.

The example database shown in Figure 1.1 is based on the schema that Facebook

User			Friend	
uid	name	hobby	uid1	uid2
1	Babbage, Charles	math	1	3
2	Church, Alonzo	math	3	1
3	Lovelace, Ada	music	2	4
4	Turing, Alan	chess	4	2
5	von Neumann, John	history	4	5
			5	4

```

CREATE VIEW V1 AS
  (SELECT name, hobby FROM User WHERE uid = 1);

CREATE VIEW V2 AS
  (SELECT name FROM User WHERE uid = 1);

CREATE VIEW V3 AS
  (SELECT U.uid, U.name FROM User U, Friend F
   WHERE U.uid = F.uid2 AND F.uid1 = 1);

```

Figure 1.1: Database and security views.

exposes to third party app developers through the Facebook Query Language (FQL) [4]. The schema contains two tables: *User*, which has a User ID (*uid*), *name*, and *hobby* for each user in the dataset, and *Friend*, which tracks friendship relations between users.

The system controls disclosure of this data using a *reference monitor* (RM) that enforces a *security policy*. At the time that the database schema is designed, a trusted system administrator defines a set of *security views* that reveal known types of information about the dataset. All subsequent policy decisions are made with the help of these security views. In Figure 1.1, V1 reveals the uid, name, and hobby of User 1 (Charles Babbage) while V2 reveals his uid and name but not his hobby and V3 reveals the uids and names of his friends. In a real system, an analogous set of security views would be generated for each end user; appropriate security views can easily be generated from templates defined by a human administrator. Many permissions used by Facebook [4] and Android [2] are essentially simple security views, though they are not formally defined in SQL.

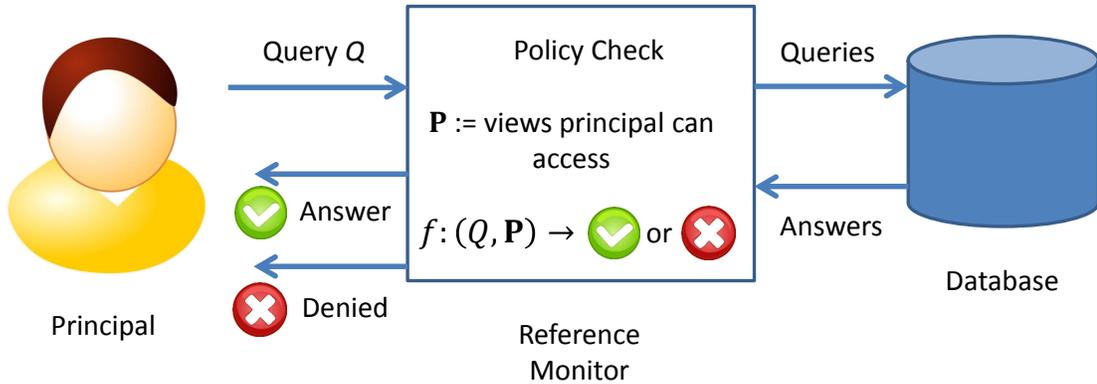


Figure 1.2: Disclosure control system model

The RM sits between the query evaluation engine and the principals who issue queries (Figure 3.2). When a principal issues a database query Q , the RM performs a policy check in order to determine whether the query should be accepted or rejected. This policy check is divided into two steps.

In the first step, the RM computes the subset \mathbf{P} of the security views to which the principal has been granted access. An untrusted third-party app running on Charles Babbage’s behalf might be permitted to see the contents of V_2 but not of V_1 or V_3 . In practice, this computation might be based on an *Access Control List* (ACL), a set of *Credentials* supplied by the issuer, or a *Role-Based* security mechanism.

In the second step, the RM checks whether the views in \mathbf{P} contain enough information to answer the principal’s original query. The determination is made by a *policy function* f that takes Q and \mathbf{P} as inputs and returns 0 or 1. If f returns 1 then the principal receives the query results; otherwise, the query is denied. The function f is said to be *database-dependent* if its output can depend on the current state of the database, or *database-independent* otherwise. Our focus in this work is on *database-independent* policies; this ensures that neither policy decisions nor the corresponding explanations

can leak potentially sensitive information about the current state of the database.

Ideally, f would be based on a simple mathematical criterion: it would determine whether the security views that the principal had access to contained enough information to answer Q . If so, f would output 1. Otherwise, it would output 0. In practice, however, f is typically a very crude and conservative algorithm.

If access control policies are specified using the GRANT and REVOKE keywords provided by the SQL standard, the onus is on the principal to write the query in terms of the views he or she can access. Let us return to Figure 1.1; suppose an app that has been granted access to V2 but not V1 issues the following query:

```
SELECT name FROM V1;
```

This query will be rejected even though it only reveals information that the app is allowed to access – specifically, it reveals the name of User 1, which can be computed from V2. The function f is not *data-derived*: it checks whether the principal is allowed to access all the views in the FROM clause, but it does not account for the information that the query and the views reveal about the underlying database.

When the query is rejected it would be helpful for the RM to point out that the query can be rewritten using V2 – which the principal has access to – in the following manner:

```
SELECT name FROM V2;
```

Better yet, the reference monitor should be able to search for rewritings automatically, and use them – if found – whenever it enforces a policy or generates an explanation.

Reference monitors that can automatically search for suitable rewritings have been proposed in previous work [46]. However, the reasons behind security policy decisions made by such reference monitors can be difficult to understand; in particular, explanations for rejected queries are often insufficient or nonexistent [42]. Even worse, systems such

as Oracle’s Virtual Private Database [11] accept unauthorized queries and modify their execution semantics without users’ knowledge [46]. In contrast, our model can provide a concise explanation for *any* policy decision, regardless of whether a query is reject or answered.

1.3 Layout of the Dissertation

The rest of this dissertation is laid out as follows. In Chapter 2 we review related work from the database and security literature. In Chapter 3 we formalize the problem of disclosure labeling and develop algorithms that can solve the problem. We provide practical results for a specific query language: conjunctive queries under set semantics. In Chapter 4 we extend the results to an extended version of this query language, and in Chapter 5 we show how the results can be extended to a fragment of SQL that includes conjunction, disjunction, correlated subqueries, NULL values, and negation. In Chapter 6 we discuss explainable security and other applications of disclosure labeling. Finally, in Chapter 7 we discuss a prototype system that was designed to assess the viability of explainable security.

The results in this dissertation were previously presented in two conference papers. The first [15] covers a subset of the material in Chapter 3. The second [14] covers a subset of the material in Chapters 5 and 6, and 7.

CHAPTER 2

RELATED WORK

In this Chapter we review prior work that is related to disclosure labeling and explainable security. In Section 2.1 we introduce the language of *conjunctive queries under set semantics*, which are central to many of the results in Chapter 3. We next review the problem of finding *rewritings* for queries in Section 2.2; these results make it possible to compare the relative amounts of information revealed by different database queries. In Section 2.3 we review work on the problem of defining and enforcing access control policies in database systems. Finally, in Section 2.4 we discuss results from the broader security literature that are relevant to this dissertation.

2.1 Queries and Views

Throughout this dissertation we will refer to *database queries* extensively. Informally, queries are functions that map datasets to tables. For example, when evaluated on the dataset shown in Figure 2.1 (which contains a single table), the SQL query

```
SELECT name FROM User WHERE uid < 7
```

will return a table with a single column whose title is “name.” The first row in this table will have the name “Alice,” and the second will have the name “Bob.”

We assume that any queries under consideration are defined in some predetermined language. The precise language we focus on will depend on the context, however: we will work with at least four different query languages in Chapters 3, 4, and 5.

Semantically, *views* and *queries* are interchangeable: they are typically defined in the same languages and have the same evaluation semantics. The difference between the

uid	name	email
1	Alice	alice@domain.net
2	Bob	bob@gmail.com
3	Charlie	charlie@hotmail.com

Figure 2.1: Example User relation.

two boils down to how they are used. Queries are issued by client applications or end users of a database. Views, on the other hand, are used internally by the DBMS to assist in the answering of queries. They can be used to integrate data from multiple sources into a unified logical schema (e.g., [38], [44]), to answer queries faster (e.g., [31]), or to enforce security policy constraints (e.g., [46], [53]).

2.2 Answering Queries using Views

In order to provide concrete examples of how views can assist in the answering of queries we next briefly introduce a concrete query language (conjunctive queries under set semantics) and review basic facts about queries in this language. Our discussion is informal; for a more rigorous discussion we refer readers to the work of Chandra and Merlin [19] or the language extension defined in Chapter 4.

2.2.1 Conjunctive Queries under Set Semantics

A conjunctive query Q takes takes the form

$$Q(\bar{t}) :- R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$$

where each R_i a relation name and each \bar{t}_i is a list of constants and variable names. We refer to $Q(\bar{t})$ as the query's *head*; intuitively it represents the query's output. We refer

```

{ User(1, 'Alice', 'alice@domain.net'),
  User(2, 'Bob', 'bob@gmail.com'),
  User(3, 'Charlie', 'charlie@hotmail.com') }

```

Figure 2.2: Representation of User table as a set of relational atoms.

to each $R_i(\bar{t}_i)$ as a *body atom*; all of the body atoms together comprise the query's *body*. Every database has an accompanying *schema*, which contains a list of relations that appear in that database and a fixed arity for each relation. For example, the schema shown in Figure 2.1 contains a relation named *User* whose arity is 3. The following conjunctive query is defined on that schema:

$$Q_1(n) :- \text{User}(4, n, e) \wedge \text{User}(u, n, e)$$

The input and output of a conjunctive query are both represented as sets of relational atoms. For example, the *User* relation shown in Figure 2.1 is represented as the set of atoms shown in Figure 2.2.

We next provide an example to illustrate how query evaluation works. The answer to the query Q_1 above consists of a set of relational atoms of the form $Q_1(n)$ where n is a constant. The atom $Q_1(n)$ appears in the query's answer if the logical predicate

$$\exists u, e . \text{User}(4, n, e) \wedge \text{User}(u, n, e)$$

is satisfied by the current dataset and does not appear in the query's answer otherwise.

Two queries are said to be *equivalent* if they return the same answer on every possible dataset. Query equivalence is characterized by *homomorphisms*. A homomorphism maps constants to themselves and maps variables to constants. A homomorphism also induces a map on relational atoms by

$$\varphi R(t_1, t_2, \dots, t_m) = R(\varphi t_1, \varphi t_2, \dots, \varphi t_m)$$

If Q and Q' are queries then a homomorphism $\varphi : Q \rightarrow Q'$ maps the head of Q to that of Q' and maps every body atom of Q to a body atom of Q' . We say that Q contains Q' , written $Q \supseteq Q'$, if the tuples in the answer to Q form a superset of those for Q' on any possible dataset. The existence of a homomorphism from Q to Q' is necessary and sufficient to prove that $Q \supseteq Q'$. Hence, the existence of homomorphisms from Q to Q' and from Q' to Q is necessary and sufficient to prove that Q and Q' return the same answer on any possible dataset.

The existence of homomorphisms is not always immediately obvious. For instance, consider the query

$$Q_2(n) :- \text{User}(4, n, e)$$

At first glance, Q_2 appears to be very different from the query Q_1 defined above. We claim, however, that the two queries will always return the same answer. By the argument above, it suffices to find homomorphisms $\varphi : Q_1 \rightarrow Q_2$ and $\varphi' : Q_2 \rightarrow Q_1$. The following homomorphisms will work:

$$\varphi = \{n \mapsto n, e \mapsto e, u \mapsto 4\}$$

$$\varphi' = \{n \mapsto n, e \mapsto e\}$$

The equivalence of Q_1 and Q_2 may at first be surprising. Intuitively, the reason is as follows. Q_1 emit the atom $Q_1(n)$ if (i) n is the name of user 4 and (ii) n is the name of at least one user in the input dataset. But if condition (i) is satisfied then condition (ii) is obviously also satisfied. Hence, the second atom of Q_1 , which corresponds to condition (ii), contributes nothing to the query's answer and can therefore be removed. The result is identical to Q_2 .

We refer to the process of removing extraneous atoms from the body of a query as *folding*. In the example above, Q_2 is a folding of Q_1 because both queries return the

same answer but Q_2 contains a proper subset of the body atoms of Q_1 . The process can be repeated iteratively until no more body atoms can be removed. The result is a *folded* query. Folding allows us to place conjunctive queries in a canonical form: if Q and Q' are equivalent and contain a minimal number of body atoms then they must be the same up to renaming of variables and reordering of body atoms. In this case we say that Q and Q' are *isomorphic*.

The problem of checking whether two queries are equivalent is known to be NP-complete. However, the complexity is expressed in terms of the sizes of the query *definitions* (which tend to be small) rather than the size of the input dataset (which tends to be large). Consequently, query equivalence can usually be checked quickly in practice.

2.2.2 Finding Query Rewritings

We previously argued that views are typically used by DBMSs to help answer queries, and this task is accomplished using *rewritings*. Rewritings are informally discussed here; we refer the interested reader to the survey of Levy et al. [38] for a more extensive discussion. An equivalent rewriting of a query Q using a set of views \mathbf{V} is a query that returns the same answer as Q but references views in \mathbf{V} rather than base relations. For example, define a view

$$V_3(u, n) :- \text{User}(u, n, e)$$

and a query

$$Q_4(n) :- V_3(4, n)$$

It is possible to show that Q_4 is an equivalent rewriting of Q_2 using V_3 . Notice that Q_4 is a deterministic function that takes the answer to V_3 as its input and returns the same answer as Q_2 as its output. Consequently, the answer to V_3 uniquely determines the

answer to Q_2 on any possible dataset. For conjunctive queries and views, the problem of determining whether Q has an equivalent rewriting using \mathbf{V} is known to be NP-complete in the sizes of the query and view definitions [38]. A practical algorithm for performing this check has been proposed by Compton [24].

In the context of security policy enforcement it is sometimes desirable to check whether the answers to a set of views \mathbf{V} determine the answer to a query Q on every possible dataset. We argued above that the existence of an equivalent rewriting of Q using \mathbf{V} is *sufficient* to ensure determinacy. However, it is not *necessary*: it is possible that \mathbf{V} determines the answer to Q on any possible dataset but no suitable rewriting exists. Unfortunately, checking determinacy directly appears to be very difficult. [41] For this reason we use the existence of an equivalent rewriting as a practical but conservative check for determinacy.

Our focus in this dissertation is on equivalent rewritings (which will often be referred to simply as “rewritings”), which are discussed above. There is also a notion of *contained rewritings* that has been widely studied by the database community but will not be used here. A *contained* rewriting Q' of Q using \mathbf{V} is a rewriting using \mathbf{V} that is only guaranteed to return a *subset* of the tuples in the answer to Q . Q' is said to be *maximally contained* if it returns a superset of the tuples that appear in any other contained rewriting. A discussion of maximally contained rewritings can be found in the survey of Halevy [33], and a practical algorithm for finding maximally contained rewritings can be found in the work of Pottinger et al. [44]

2.2.3 Query Pricing

The notion of determinacy discussed above was *dataset-independent*: our aim was to determine whether the answer to a query was uniquely determined by the answers to a set of views on *every* possible dataset. In contrast, Koutris et al. [35] consider an *instance-based* notion of view determinacy; their goal is to determine whether the answers to a set of views determine the answer to a given query on a *specific* dataset. To understand the distinction, consider the views

$$V_5(u, n, e) :- \text{User}(u, n, e)$$

$$V_6() :- \text{User}(u, n, e)$$

In general, the answer to V_6 does *not* determine the answer to V_5 , since the latter reveals information about all the columns of the `User` table whereas the former does not reveal information about any of them. Suppose, however, that the `User` table is empty. In this case determinacy *does* hold: by looking at the answer to V_6 we can infer that the `User` table is empty, and therefore the answer to V_5 must also be empty.

Koutris et al. relate instance-based determinacy to the problem of pricing queries in *data markets*. In this setting, a vendor defines a set of views \mathbf{V} over a dataset and assigns a price $\pi(V)$ to each view $V \in \mathbf{V}$. Given a query Q , their goal is to find a subset of the views in \mathbf{V} that allows the answer to Q to be computed as cheaply as possible. More precisely, they require that (i) the answer to the views in \mathbf{V} uniquely determine the answer to Q on the current dataset, and (ii) the sum of the prices of the views in \mathbf{V} is as small as possible. A naïve realization of the second criterion would require a brute-force search over all possible subsets of the views in \mathbf{V} , but the check can often be performed more efficiently in practice.

2.3 Database Access Control

We now shift our attention to database access control mechanisms, which allow a trusted administrator to restrict which parts of a database each principal can read and write; our focus here is on mechanisms that can restrict the information that principals can *read*.

In an ideal world, every security principal in a system would be granted access to some subset of the information in the database. The term “information” in this context is ambiguous. We might regulate access to information at the granularity of *tables* in a database. In this model, a principal can either read all the information in a given table or cannot read any of it. This is the approach taken by the GRANT and REVOKE keywords in the SQL standard. An administrator who wishes to grant a principal named *Alice* access to the *User* table shown in Figure 2.1 will issue the following command:

```
GRANT SELECT ON User TO Alice
```

The administrator can revoke Alice’s access to the *User* table by typing

```
REVOKE SELECT ON User FROM Alice
```

Table-level access control mechanisms are relatively straightforward to reason about for both humans and computers, and are widely adopted in practice. Support for the GRANT and REVOKE keywords is provided by many database management systems (DBMSs), including PostgreSQL [6], MySQL [7], Oracle Database [5], and MS SQL Server [8].

The main disadvantage of using table-level access control is its coarse granularity: it is often desirable to grant a principal access to only a *subset* of the information in a table. *Row-Level* access control mechanisms such as IBM’s *Label-Based Access Control* [9] can be used to grant a principal access to a restricted subset of the rows in a table. For instance, an administrator might wish to permit Alice to see information about herself but not about any other users in the dataset. Similarly, *Column-Level* access control

mechanisms can be used to grant a principal access to a column of the rows in a table. For instance, we might permit Alice to see *uids* and *names* but not e-mail addresses. In [20], Chaudhuri et al. propose an extension of the SQL GRANT and REVOKE keywords that supports both row-level and column-level access control.

Both row-level security and column-level security are essentially subsumed by *view-based* security. In the view-based security model, an administrator grants each principal access to a set of *views* over a dataset. A principal is permitted to read the contents of the views he is granted access to. Consequently, a principal can evaluate any query whose answer is uniquely determine by these views.

View-based security mechanisms can simulate both row-level security and column-level security. Row-level security constraints can be simulated by views with *selection conditions*. For instance, we can grant Alice access to her own information (but nobody else's) with the view

```
CREATE VIEW V7 AS
SELECT * FROM User WHERE uid = 1
```

Similarly, column-level security constraints can be simulated by views with *column projections*. For example, we can grant Alice access to the uid and name columns by allowing her to read the view

```
CREATE VIEW V8 AS
SELECT uid, name FROM User
```

The SQL GRANT and REVOKE keywords permit a very simple version of view-based security: a principal can be granted access to views as well as tables, and is permitted to execute any query that references only view that he has been granted access to. A principal who was granted access to the view V_8 could execute the query

```
SELECT name FROM V8
```

but would not be permitted to execute the query

```
SELECT name FROM User
```

even though both queries are guaranteed to return the same answer.

Security mechanisms based on *view rewriting* take this idea one step further. Rather forcing a principal to write the query above in terms of V_8 as the SQL GRANT keyword does, such mechanisms can check for such rewritings *automatically*. Such mechanisms can broadly be divided into two categories: *Truman* and *non-Truman*. In both cases, the goal is to allow principals to execute authorized queries while preventing them from executing unauthorized queries. More precisely, we strive to answer as many queries as possible while ensuring that any information that a principal learns about the dataset from the queries' answers could have been inferred simply by looking at the views that he was granted access to.

However, the two methods differ in their handling of unauthorized queries. A *non-Truman* mechanism will either allow a query to be executed *as written* or will reject it outright. In contrast, a *Truman* mechanism will semantically modify the query prior to execution to ensure that its answer depends only on information that the principal is permitted to access.¹ We next discuss both approaches in greater detail. In the discussion below we assume that the database schema is publicly known, whereas the actual contents of the database are potentially sensitive and must be protected.

¹The terminology is a reference to Truman Burbank, the title character of the 1998 movie *The Truman Show*. [46] Truman is provided with the illusion of freedom, but his environment is carefully and deliberately manipulated to ensure that he never leaves the island town where he lives. In a somewhat tenuous analogy, a principal in the Truman model is provided with the illusion of unrestricted access to a database, but its queries are carefully and deliberately manipulated to enforce an administrator-defined security policy.

2.3.1 Non-Truman View Rewriting

In the non-Truman model, a principal can be assured that he will either receive a *correct* answer to his query or will not receive any answer at all. For example, we argued above that if a principal who is granted access only to V_8 issues the query

```
SELECT name FROM User
```

can safely be answered as is. After all, the answer to V_8 determines the query's answer on *any possible database*. Using the language of Rizvi et al. [46], a query is said to be *unconditionally valid* in this case. On the other hand, the query

```
SELECT * FROM User
```

usually *cannot* be answered using V_8 alone, since it depends on *all* the columns of the `User` relation whereas V_8 only reveals information about *uids* and *names*. There is one subtle but important exception to this rule: if the answer to V_8 is empty then the `User` relation must also be empty, so the answer to the query above must be empty as well. In this case V_8 *does* determine the query's answer, at least on the current dataset. In Rizvi's terminology, a query is said to be *conditionally valid* in this case. The distinction between unconditional and conditional validity mirrors the distinction between dataset-independent and instance-based view rewriting from Section 2.2.3

A serious disadvantage of non-Truman view rewriting (at least in the form proposed by Rizvi) is its apparent intractability. Determining whether a SQL query is unconditionally valid is known to be undecidable [46], and we are not aware of any algorithms that can decide whether a query is conditionally valid. Rizvi et al. provide inference rules that are *sound* but not *complete*: every query that is answered will be conditionally valid, but some conditionally valid queries might be rejected by the system. A second disadvantage is that policy decisions can be opaque: when a query is rejected, it can be difficult for users to understand why. [42]

2.3.2 Truman View Rewriting

In the Truman model, a principal's query's will *always* be answered. However, queries may be semantically modified prior to execution. For instance, in the model proposed by Olson et al. [42] the query

```
SELECT * FROM User
```

would be silently rewritten as

```
SELECT uid, NULL AS name, NULL as email
```

prior to execution. The result will be a table containing three columns named “uid,” “name” and “email.” However, the “name” and “email” columns will always be NULL. Security mechanisms that semantically modify queries prior to execution have been deployed in popular commercial DBMSs; notable examples include Oracle's *Virtual Private Database* [11] and IBM's *Label-Based Access Control* [9]. There is also a substantial body of academic research dedicated to Truman view rewriting, including Olson et al.'s *Reflective Database Access Control* [42], the *Predicated Grant* model proposed by Chaudhuri et al. [20], and LeFevre et al.'s proposal for access control in Hippocratic databases [37].

Truman view rewriting offers the promise of separating security policy enforcement from query formulation. A principal can issue queries as if he had unrestricted access to the entire database, and the system will silently rewrite his queries to ensure that he never learns more information than he is supposed to. However, the query that a principal issues may be very different from the query that is finally answered by the DBMS, and this discrepancy can lead to query answers that are misleading or blatantly wrong. [46] If a principal who is only granted access to V_7 tries to execute the query

```
SELECT COUNT(*) FROM User
```

then his query will be silently rewritten to

```
SELECT COUNT(*) FROM User WHERE uid = 1
```

prior to execution and will therefore provide an absurdly low estimate of the number of users in the database.

A principled treatment of Truman view rewriting is provided by Wang et al. [53], who identify and formalize three desirable properties for reference monitors that employ Truman view rewriting:

- (i) **Sound:** The rewritten query should return only *correct* information.
- (ii) **Secure:** The rewritten query should depend only on information that is permitted by the security policy.
- (iii) **Maximum:** The query should return as much information as possible.

They argue that Truman view rewriting should be and complete but, due to practical considerations, should only strive to be maximal.

2.3.3 Inference Control

View-based security mechanisms typically place an *upper bound* on the information that a principal *is* allowed to access. Any administrator defines a set of views that a principal is permitted to read, and the system ensures that any information that the principal learns by issuing queries could also have been learned by looking at the views he is granted access to. In contrast, mechanisms for *privacy-preserving data publishing* place a *lower bound* on the information that a principal will *not* learn by looking at a query's answer.

In a typical setup, a hospital might wish to permit an academic researcher to issue

aggregate queries over a sensitive medical dataset while preventing him from learning too much information about any specific individual in that dataset. A number of techniques for privacy-preserving data publishing have been proposed in the academic literature, including k -anonymity [52], ℓ -diversity [39], and ϵ -differential privacy [26]. However, there is a very concrete sense in which privacy-preserving data publishing is more difficult than access control: access control mechanisms typically do not make any assumptions about a principal's background knowledge, while privacy-preserving data publishing must. [26, 34]

ϵ -differential privacy is a popular criterion that ensures that an adversary who is not aware of any correlations between different tuples in a dataset will learn no more than $(\epsilon \cdot \log_2 e)$ bits of information about any single tuple by seeing the output of a published query. [22] Differential privacy is often achieved by injecting random noise into a query answer; a smaller value of ϵ will lead to stronger privacy guarantees; however, these guarantees will be typically achieved by making query answers noisier.

Differential privacy aims to limit the amount of information that is revealed about *any* tuple in a dataset. *Perfect privacy* [40] differs from differential privacy in two ways. First, it provides guarantees only for a distinguished subset of the tuples in the dataset. And second, it ensures that a query will not reveal *any* information at all about the tuples in this distinguished subset. Another form of inference control is proposed by Brodsky et al. [18]. Their algorithm is able to infer a list of potentially sensitive facts that could be learned from the answers to a collection of database queries.

2.4 Security and Privacy Potpourri

We next discuss work from the security and privacy communities that is relevant to this dissertation but is not targeted towards database systems in particular.

2.4.1 Programming Language Security

Both Truman and non-Truman view rewriting have parallels in the programming language security literature. Non-Truman view rewriting can be thought of as a form of static program analysis: in both cases, the goal is to determine through static analysis whether a program is safe to run. Similarly, Truman view rewriting has parallels with Inlined Reference Monitors, which rewrite untrusted programs into *safe* ones that are guaranteed to satisfy certain security properties. For a discussion of both techniques we refer the interested reader to the survey of Schneider et al. [49]

Another innovation from the programming language security literature that is relevant to this dissertation is Landauer et al.'s *lattice of information* [36]. Roughly speaking, they introduce a set of functions that map program states to an arbitrary output domain, and orders these functions based on the amount of information that they reveal. These functions induce a lattice. The top element of this lattice corresponds to the identity function, which completely reveals a program's current state. The bottom element of this lattice corresponds to a constant function, which reveals no information about a program's current state. The order can formally be defined in terms of function composition: $f \leq g$ if there is a function h such that $f = h \circ g$. In other words, any information that could be learned by looking at the answer to f could also be learned by looking at the answer to g and performing appropriate postprocessing. Although Landauer et al.'s formalism

is closely related to results on disclosure lattices in Chapter 3 of this dissertation, their results are primarily theoretical, while we aim to reason about information disclosure in real query languages.

Lattice-based models can also be used to encode and help enforce practical security constraints. Roughly speaking, the goal of *information flow security* is to ensure that a program which operates on both sensitive and non-sensitive information will not allow sensitive inputs to impact publicly visible outputs. There is an extensive body of work on the topic; we refer interested readers to the survey of Sabelfeld et al. [47] One key difference from our technique is that the lattices used by information flow control systems are typically not data-derived.

2.4.2 Reasoning about Security Policies

Most of the models discussed above assume that a security policy is defined by a single entity. However, this will not always be the case. In practice, security policies in distributed environments must balance the concerns of multiple entities with disparate interests. A number of formalisms have been proposed to handle such cases.

Bonatti et al. [16] propose an algebra with a number of operations for composing different access control policies. Authorization logics such as SecPAL [13] and NAL [50] permit rich reasoning about security policy decisions.

2.4.3 Usability of Security Permissions

The work we have discussed up until this point focuses the problem of *enforcing* security policies. In practice, however, the humans who specify these policies can and do make mistakes. In fact, much of the motivation for this dissertation is based on the assertion that users and developers struggle to use complex permissions structures. We next review two studies of Google's Android by Felt et al. [28, 29] that support this claim.

In [28], Felt et al. developed a tool that instrumented Android apps to determine what API calls they made. They used these API calls to determine what permissions apps made use of when they ran, and were able to identify permissions that apps requested but never actually used. They ran their tool on 900 apps, and it identified 323 (35.8%) as having erroneous permissions; however, they may have had a high false positive rate due to limitations of their tool. In a sample of 40 apps that were reviewed by hand they identified 17 out of the 18 apps flagged by their tool as true positives and the remaining app as a false positives.

They identified a number of causes for overprivileged apps, some of which are reviewed below:

- (i) **Permission name:** Developers requested permissions that looked relevant to their apps' functionality but were not actually required. For instance, developers who needed the `ACCESS_NETWORK_STATE` permission would frequently request the `ACCESS_WIFI_STATE` even if it wasn't used, and vice versa; they speculated that this was because the permissions had similar-sounding names.
- (ii) **Deputies:** One Android app can invoke another (the *deputy*) with a request to perform a specific operation. For example, third-party apps can launch the Android camera app as a deputy. In this case the `CAMERA` permission is needed by the

camera app to snap pictures but is not needed by the third-party app; however, developers sometimes requested it anyway.

- (iii) **Related Methods:** Within a single API, some functions might require permissions that others do not. This appeared to confuse some developers.
- (iv) **Buggy Documentation:** Both the official documentation and unofficial message boards can contain erroneous information about Android permissions, and developers who rely on these sources may request the wrong permissions as a result.
- (v) **Testing Artifacts:** Developers sometimes requested certain permissions (such as `ACCESS_MOCK_LOCATION`) that were only needed during testing but neglected to remove them when their apps were released.

Do the problems that Felt et al. observed on the Android platform apply to other systems? We don't know for certain, but the causes of overprivileged apps listed above are hardly unique to Android. For example, in Section 3.6.1 we show that the API that Facebook exposes to third-party contains a large number of permissions, some of which are confusingly named and some of which are poorly or incorrectly documented.

In a second study Felt et al. [29] attempted to gauge user comprehension of Android permissions through the combination of a lab study and an Internet survey. Prior to installing an app, Android presents the end user with a list of permissions that the app must be granted before it can be run. However, Felt et al. found that users often paid little attention to the permissions that apps requested. Furthermore, many users did not understand what operations were enabled by specific permissions. However, a minority of study participants demonstrated a good understanding of the Android permission system. Such expert users can influence the behavior of others by writing app reviews in which they point out suspicious permission requests.

CHAPTER 3

DISCLOSURE LABELING

In this Chapter we introduce *disclosure labeling*, which provides a formal foundation for reasoning about information disclosure in database systems that is data-derived, semantically meaningful and expressive. Our notion of disclosure is based on the idea of associating levels of disclosure with views over the database. Some views disclose more information than others; this induces a hierarchy. For example, consider a user Alice whose calendar and contacts data is shown in Figure 3.1 (a). The view V_1 contains information from the full Meeting table, while V_2 displays only the time slots of appointments. Clearly V_2 reveals less information than V_1 .¹

Although apps may ask arbitrary queries on the data, Alice can formulate a policy based on a small set of *security views* such as those in Figure 3.1 (b). For example, she may specify that she is happy to disclose V_2 but not V_1 ; therefore, any query that can be answered using only information in V_2 is permitted but queries requiring more information are forbidden. To enforce this policy, every query Q is *automatically* associated (“labeled”) with the set of security views that is required to answer Q but reveals as little information as possible beyond that. For example, the label of Q_1 in Figure 3.1 (c) is $\{V_1\}$ and the label of Q_2 is $\{V_1, V_3\}$. Policies are defined in terms of labels; Alice can specify that any query whose label is just $\{V_2\}$ can be answered, but queries with labels that are “above” (more informative than) V_2 should be rejected. Both Q_1 and Q_2 would be rejected under such a policy.

Figure 3.2 illustrates the full workflow. The user, perhaps with assistance by the

¹Many app ecosystems, including the iOS and Android mobile platforms, contain such data, and this type of data is typically considered sensitive by users [27, 30]. In fact, LinkedIn came under fire in 2012 for writing an iOS app that sent sensitive information extracted from users’ calendars back to LinkedIn’s servers [43].

Meeting		Contact		
Time	Person	Person	Email	Position
9	Jim	Jim	jim@e.com	Manager
10	Cathy	Cathy	cathy@e.com	Intern
12	Bob	Bob	bob@e.com	Consultant

(a) Dataset

$V_1(x, y) :- \text{Meeting}(x, y)$
 $V_2(x) :- \text{Meeting}(x, y)$
 $V_3(x, y, z) :- \text{Contact}(x, y, z)$

(b) Security Views

$Q_1(x) :- \text{Meeting}(x, 'Cathy')$
 $Q_2(x) :- \text{Meeting}(x, y) \wedge \text{Contact}(y, w, 'Intern')$

(c) Queries

Figure 3.1: Sample dataset, security views, and queries.



Figure 3.2: Disclosure control system model

platform developer and/or third party privacy watchdog groups, creates a base set of security views. Upon this set of views, the user defines a security policy that specifies what level of disclosure is permitted; again, other parties could help by pre-defining sensible policies which users could adjust as desired. The disclosure restriction is enforced by a *reference monitor* – a component that automatically computes the labels for all incoming queries, and accepts or rejects queries to ensure that the security policy is never violated. Figure 3.2 is a conceptual diagram rather than an architectural specification. In practice, the reference monitor could be an independent system component or a part of the DBMS or embedded within the untrusted app.

The rest of this Chapter is laid out as follows. We review relevant notation in Section

3.1 and formalize the problem of disclosure labeling in Section 3.2. Our model of information disclosure satisfies the desiderata identified in Chapter 1. It is data-derived because the labeler associates each query directly with the information required to answer it. And it is semantically meaningful because each query is labeled with a set of views that concisely characterize the information that it discloses.

Next, in Section 3.3 and 3.4 we provide practical algorithms for labeling conjunctive queries under set semantics. In Section 3.5 we describe a prototype system that we built to evaluate our algorithms and discuss additional optimizations that were integrated into the prototype. In contrast with many conventional view-based security mechanisms, our prototype is able to support *stateful* policies such as Chinese Wall policies² in which the decision to answer or reject on the query may depend on the queries that a principal has issued in the past.

Finally, in Section 3.6 we apply our framework to practical disclosure control scenarios. As a case study, we review Facebook’s hand-crafted permissions labeling of FQL and Graph API queries and discover multiple inconsistencies and problems with the documentation. We also show that our algorithms scale well on realistic workloads (Section 3.6).

3.1 Problem Statement

The goal of disclosure labeling is to determine what information about the underlying database is disclosed by answering an arbitrary set of queries Q . Labeling makes use of

²*Chinese Wall* security policies, which are prevalent in the commercial world, are used to mitigate conflicts of interest that might arise if a firm has multiple clients with competing interests. For instance, we might wish to prevent an analyst who has previously received confidential information about Microsoft from subsequently consulting for a competitor such as Google. [17] In the context of database security, we wish to prevent a principal whose queries have previously accessed sensitive information about Microsoft from subsequently issuing queries that retrieve sensitive information about Google, and vice versa.

a set \mathbf{S} of *security views* that reveal known and semantically meaningful types of information about the database. A *disclosure labeler* is a function that relates the information revealed by \mathbf{Q} to the information revealed by the security views in \mathbf{S} . Specifically, it identifies a subset of \mathbf{S} which is sufficient to answer all the queries in \mathbf{Q} , but otherwise discloses as little additional information as possible.

Disclosure labelers provide a formal foundation for the disclosure control in app ecosystems which is our ultimate goal. In addition, they highlight previously unexplored connections between view-based security and order theory.

Defining disclosure labelers and providing labeling algorithms are both challenging. A precise definition of disclosure labeling requires us to formalize the concept of disclosing “as little information as possible” about the database. Existing theoretical work [36] addresses this to an extent, but has limited practical applicability.

In the remainder of this section, we situate disclosure labeling within a broader problem space; we present our solution to the above challenges starting in Section 3.2.

3.1.1 Notation and Terminology

We close this section by introducing a few additional pieces of terminology and notation. Although Sections 3.2 and 3.3 do not assume any particular query language, we will later restrict our attention to conjunctive queries over the schema of a fixed database D . A conjunctive query has the form

$$H :- B$$

where H is a relational atom and B a conjunction of relational atoms over database relations. H and B are the *head* and *body* of the query, respectively. Each atom may

contain constants and variables. Any variables that appear in H must also appear in B . Letters x, y, z etc. indicate variables and letters a, b, c etc. indicate constants. A *distinguished* variable is one that appears in the head of the query, and an *existential* variable is one that appears only in the body. We say that two queries are *equivalent* if they return the same answer on every dataset.

Section 3.2 formalizes disclosure in terms of preorders and lattices. Given a set C , a binary relation is a subset of $C \times C$. A relation \sim is *reflexive* if $c \sim c$ for all $c \in C$; it is *symmetric* if $c \sim c'$ if and only if $c' \sim c$; it is *antisymmetric* if $c \sim c'$ and $c' \sim c$ together imply $c = c'$; and it is *transitive* if $c \sim c'$ and $c' \sim c''$ together imply $c \sim c''$. A *preorder* is a binary relation that is reflexive and transitive. An *equivalence relation* is a preorder that is also symmetric. A *partial order* is a preorder that is also antisymmetric. If \leq is a partial order, then C forms a *lattice* under \leq if every pair of elements from C has both a least upper bound (LUB) and greatest lower bound (GLB). A *bounded lattice* contains a least element \perp and a greatest element \top ; all lattices we consider are bounded.

3.2 Disclosure Labeling

This section introduces a formal framework for measuring and controlling disclosure that draws on fundamental connections between view-based security and order theory. *Disclosure orders* (Section 3.2.1) and *disclosure lattices* (Section 3.2.2) allow us to reason about the amounts of information disclosed by different set of views. We formally define *disclosure labelers* in Section 3.2.3. It turns out not every set of security views is suitable for creating a disclosure labeler; we give conditions that guarantee the existence and uniqueness of disclosure labelers. In Section 3.2.4, we explain the conceptual end-to-end setup for the use of disclosure labeling in disclosure control, including a formal definition

Notation	Description
\mathbf{U}	universe of all possible queries
\mathbf{S}	set of security views, $\mathbf{S} \subseteq \mathbf{U}$
$\mathbf{V}, \mathbf{V}_1, \mathbf{V}_2$	sets of views; $\mathbf{V}, \mathbf{V}_1, \mathbf{V}_2 \subseteq \mathbf{U}$
V, V_1, V_2	views; $V, V_1, V_2 \in \mathbf{U}$
Q, Q_1, Q_2	queries (to be labeled); $Q, Q_1, Q_2 \in \mathbf{U}$
\leq	disclosure order
$(\Downarrow \mathbf{V})$	set of all views below \mathbf{V} under \leq
\mathcal{I}	elements of the disclosure lattice over \mathbf{U} induced by \leq
$\wp(\mathbf{U})$	power set of \mathbf{U} , i.e. collection of all subsets of \mathbf{U}
\mathcal{F}	set of disclosure labels, where each label is a set of views; $\mathcal{F} \subseteq \wp(\mathbf{U})$
ℓ	a disclosure labeler
$\ell(\mathcal{I})$	all elements of the lattice of disclosure labels for ℓ
\mathcal{P}	a security policy; can be represented as a subset of $\ell(\mathcal{I})$

Table 3.1: Notation summary

of a security policy. Our notation is summarized in Table 3.1.

3.2.1 Disclosure Orders

We now define *disclosure orders*, which formalize the notion that some sets of views disclose more information than others. Assume all views are drawn from a finite universe \mathbf{U} . A disclosure order \leq ranks the relative information revealed by different sets of views. Roughly speaking, $\mathbf{V}_1 \leq \mathbf{V}_2$ means that all the information revealed by \mathbf{V}_1 is also revealed by \mathbf{V}_2 .

A natural candidate for such an order is based on *view determinacy* [41]. Under this order, $\mathbf{V}_1 \leq \mathbf{V}_2$ if and only if the answers to all the views in \mathbf{V}_1 are uniquely determined

by the answers to the views in \mathbf{V}_2 . Unfortunately, checking this criterion is highly intractable for many classes of queries. *Equivalent view rewriting* provides a conservative approximation to the determinacy ordering in which $\mathbf{V}_1 \leq \mathbf{V}_2$ precisely when, for each view $W \in \mathbf{V}_1$, there exists an equivalent rewriting of W in terms of the views in \mathbf{V}_2 . In contrast to determinacy, equivalent view rewriting is known to be tractable for many classes of queries [38].

The choice of disclosure order will depend on multiple factors: required throughput, sensitivity to false negatives, and the complexity of the query language under consideration. Rather than restrict ourselves to one of the orders defined above, we develop a more general framework that is applicable to *any* binary relation which satisfies three basic properties. The first states that adding new elements to a set of views can only increase the amount of information that it reveals about the database. The second is a standard transitivity condition. The third allows us to derive meaningful upper bounds on information disclosure even for adversaries who combine information from multiple sources.

Definition 3.2.1. A *disclosure order* is a binary relation \leq on subsets of \mathbf{U} such that

- (i) If $\mathbf{V}_1 \subseteq \mathbf{V}_2$ then $\mathbf{V}_1 \leq \mathbf{V}_2$.
- (ii) If $\mathbf{V}_1 \leq \mathbf{V}_2$ and $\mathbf{V}_2 \leq \mathbf{V}_3$ then $\mathbf{V}_1 \leq \mathbf{V}_3$.
- (iii) If $\mathbf{V}_1 \leq \mathbf{V}_3$ and $\mathbf{V}_2 \leq \mathbf{V}_3$ then $\mathbf{V}_1 \cup \mathbf{V}_2 \leq \mathbf{V}_3$.

Both orders mentioned above are disclosure orders, but others exist too. One example is the usual set order, where $\mathbf{V}_1 \leq \mathbf{V}_2$ precisely when $\mathbf{V}_1 \subseteq \mathbf{V}_2$.

Disclosure orders need not be partial orders, as they are in general *not* antisymmetric.

For example, consider the following two views on **Meeting**, abbreviated as **M**.

$$V_1(x, y) :- \mathbf{M}(x, y) \qquad V'_1(y, x) :- \mathbf{M}(x, y)$$

V_1 and V'_1 each disclose all of **M**, so $\{V_1\} \leq \{V'_1\}$ and $\{V'_1\} \leq \{V_1\}$ under determinacy and equivalent view rewriting, but the two sets are clearly not equal. Despite being unequal, $\{V_1\}$ and $\{V'_1\}$ reveal equivalent information about **M**, since each set can be computed from the other. We can show, however, that such sets induce an equivalence relation.

Proposition 3.2.2. *The binary relation defined by $\mathbf{V}_1 \equiv \mathbf{V}_2$ if and only if $\mathbf{V}_1 \leq \mathbf{V}_2$ and $\mathbf{V}_2 \leq \mathbf{V}_1$ is an equivalence relation.*

Proof. We verify each of the properties of an equivalence relation in turn:

- (i) **Reflexivity:** Follows immediately from the reflexivity of \leq .
- (ii) **Symmetry:** $(\mathbf{V}_1 \equiv \mathbf{V}_2) \iff (\mathbf{V}_1 \leq \mathbf{V}_2 \text{ and } \mathbf{V}_2 \leq \mathbf{V}_1) \iff (\mathbf{V}_2 \equiv \mathbf{V}_1)$.
- (iii) **Transitivity:** Suppose $\mathbf{V}_1 \equiv \mathbf{V}_2$ and $\mathbf{V}_2 \equiv \mathbf{V}_3$. Then $\mathbf{V}_1 \leq \mathbf{V}_2 \leq \mathbf{V}_3$ and $\mathbf{V}_3 \leq \mathbf{V}_2 \leq \mathbf{V}_1$. It follows that $\mathbf{V}_1 \equiv \mathbf{V}_3$. □

3.2.2 Disclosure Lattices

Before we move to labeling, there are two more foundational questions to address. First, given two sets of views \mathbf{V}_1 and \mathbf{V}_2 , what information is disclosed to someone who knows both of them beyond what could have been inferred from just one of the two sets? This is the classical problem of *information combination*. Second, what common information, or *overlap*, is there in the information revealed by the two sets of views?

One might be tempted to use the union and intersection of \mathbf{V}_1 and \mathbf{V}_2 to answer the above questions. However, intersection does not work as a measure of overlap. To see

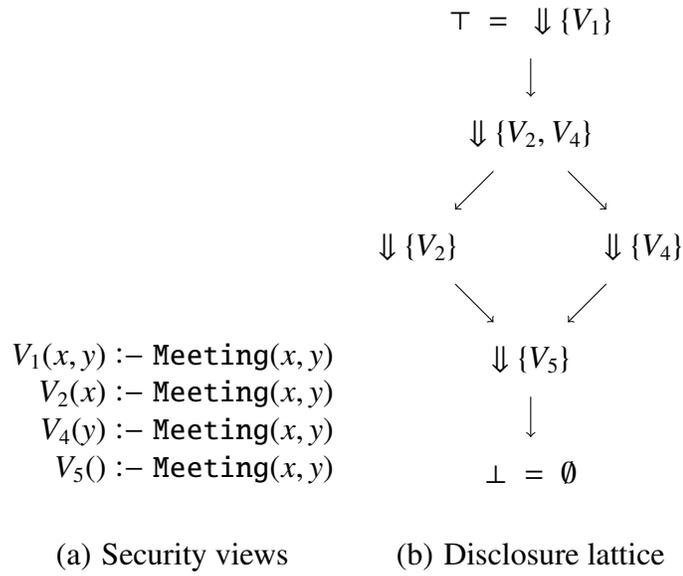


Figure 3.3: Security views and corresponding disclosure lattice.

this, consider the views V_2 and V_4 in Figure 3.3(a). The sets $\{V_2\}$ and $\{V_4\}$ have an empty intersection, but they do have some overlap. Given the answer to either V_2 or V_4 , it is possible to deduce whether `Meeting` is nonempty, i.e. to answer the query V_5 .

To define combination and overlap, we introduce a new operator ($\Downarrow \mathbf{V}$) that returns all the views in \mathbf{U} whose answers can be inferred by observing a set of views \mathbf{V} . This more accurately represents all the information disclosed by \mathbf{V} .

Definition 3.2.3. *Let $\mathbf{V} \subseteq \mathbf{U}$. Then we define $(\Downarrow \mathbf{V})$ to be $\{V \in \mathbf{U} : \{V\} \leq \mathbf{V}\}$.*

We next prove some basic properties of $(\Downarrow \mathbf{V})$.

Proposition 3.2.4. *Let $\mathbf{V}_1, \mathbf{V}_2 \subseteq \mathbf{U}$. The following are equivalent:*

- (i) $\mathbf{V}_1 \leq \mathbf{V}_2$.
- (ii) $\mathbf{V}_1 \subseteq (\Downarrow \mathbf{V}_2)$.
- (iii) $(\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_2)$.

Proof. We prove each of the following implications in turn:

(i) *implies* (ii): Suppose $V \in \mathbf{V}_1$. Then $\{V\} \subseteq \mathbf{V}_1$, so that $\{V\} \leq \mathbf{V}_1$. Since $\mathbf{V}_1 \leq \mathbf{V}_2$, and \leq is transitive, it must be the case that $\{V\} \leq \mathbf{V}_2$. It follows that $V \in (\Downarrow \mathbf{V}_2)$.

(ii) *implies* (i): Since $\mathbf{V}_1 \subseteq (\Downarrow \mathbf{V}_2)$, we know that $\{V\} \leq \mathbf{V}_2$ for each $V \in \mathbf{V}_1$. It follows that $\mathbf{V}_1 = \bigcup_{V \in \mathbf{V}_1} \{V\} \leq \mathbf{V}_2$.

(i) *implies* (iii): Take any $V \in (\Downarrow \mathbf{V}_1)$. Then $\{V\} \leq \mathbf{V}_1$, and therefore $\{V\} \leq \mathbf{V}_2$ because \leq is transitive. It follows that $V \in (\Downarrow \mathbf{V}_2)$.

(iii) *implies* (i): Suppose $V \in \mathbf{V}_1$. Then $\{V\} \subseteq \mathbf{V}_1$, and therefore $\{V\} \leq \mathbf{V}_1$, so that $V \in (\Downarrow \mathbf{V}_1)$. This means that $V \in (\Downarrow \mathbf{V}_2)$, so that $\{V\} \leq \mathbf{V}_2$. It follows that $\mathbf{V}_1 = \bigcup_{V \in \mathbf{V}_1} \{V\} \leq \mathbf{V}_2$. □

Corollary 3.2.5. *If $\mathbf{V}_1 \subseteq \mathbf{V}_2$ then $(\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_2)$.*

Proof. If $\mathbf{V}_1 \subseteq \mathbf{V}_2$ then $\mathbf{V}_1 \leq \mathbf{V}_2$, and therefore $(\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_2)$. □

Using the properties identified above, we can show that $(\Downarrow \mathbf{V})$ is idempotent.

Corollary 3.2.6. *If $\mathbf{V} \subseteq \mathbf{U}$ then $(\Downarrow (\Downarrow \mathbf{V})) = (\Downarrow \mathbf{V})$*

Proof. Let $\mathbf{V}_1 = \mathbf{V}_2 = (\Downarrow \mathbf{V})$. Then $(\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_2)$ implies $\mathbf{V}_1 \subseteq (\Downarrow \mathbf{V}_2)$, and therefore $(\Downarrow \mathbf{V}) \subseteq (\Downarrow (\Downarrow \mathbf{V}))$.

On the other hand, if $V \in (\Downarrow \mathbf{V})$ then $\{V\} \leq \mathbf{V}$, and so $(\Downarrow \mathbf{V}) = \bigcup_{V \in (\Downarrow \mathbf{V})} \{V\} \leq \mathbf{V}$. Let $\mathbf{V}_1 = (\Downarrow \mathbf{V})$ and let $\mathbf{V}_2 = \mathbf{V}$. Then $\mathbf{V}_1 \leq \mathbf{V}_2$ implies $(\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_2)$, and therefore $(\Downarrow (\Downarrow \mathbf{V})) \subseteq (\Downarrow \mathbf{V})$.

Since $(\Downarrow \mathbf{V}) \subseteq (\Downarrow (\Downarrow \mathbf{V}))$, and $(\Downarrow (\Downarrow \mathbf{V})) \subseteq (\Downarrow \mathbf{V})$, the two must be equal. □

We next verify that \mathbf{V} and $\Downarrow \mathbf{V}$ reveal the same information about the dataset.

Corollary 3.2.7. $\mathbf{V} \equiv \Downarrow \mathbf{V}$.

Proof. $\mathbf{V} \leq \mathbf{V}$ implies that $\mathbf{V} \subseteq (\Downarrow \mathbf{V})$, and therefore $\mathbf{V} \leq (\Downarrow \mathbf{V})$ as well. On the other hand, $\{V\} \leq \mathbf{V}$ for each $V \in (\Downarrow \mathbf{V})$, so that

$$(\Downarrow \mathbf{V}) = \bigcup_{V \in \Downarrow \mathbf{V}} \{V\} \leq \mathbf{V}$$

It follows that $\mathbf{V} \equiv (\Downarrow \mathbf{V})$ as desired. \square

If \mathbf{V}_1 and \mathbf{V}_2 reveal the same information about the dataset then $\Downarrow \mathbf{V}_1$ and $\Downarrow \mathbf{V}_2$ must be equal. In this way, the \Downarrow operator partitions \mathbf{U} into equivalence classes; two sets are in the same equivalence class if any view whose answer can be inferred from the first set can also be inferred from the second set.

Corollary 3.2.8. $\mathbf{V}_1 \equiv \mathbf{V}_2$ if and only if $(\Downarrow \mathbf{V}_1) = (\Downarrow \mathbf{V}_2)$.

Proof.

$$\begin{aligned} & \mathbf{V}_1 \equiv \mathbf{V}_2 \\ \iff & \mathbf{V}_1 \leq \mathbf{V}_2 \text{ and } \mathbf{V}_2 \leq \mathbf{V}_1 \\ \iff & (\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_2) \text{ and } (\Downarrow \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_1) \\ \iff & (\Downarrow \mathbf{V}_1) = (\Downarrow \mathbf{V}_2) \end{aligned} \quad \square$$

The \Downarrow operator now allows us to formalize information combination and overlap. Given \mathbf{V}_1 and \mathbf{V}_2 , the information that can be derived from both \mathbf{V}_1 and \mathbf{V}_2 taken together is $\Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2)$, and the information overlap of \mathbf{V}_1 and \mathbf{V}_2 is $(\Downarrow \mathbf{V}_1) \cap (\Downarrow \mathbf{V}_2)$. We can use these functions to construct a lattice structure that precisely captures the information disclosed by each subset of \mathbf{U} .

Theorem 3.2.9. *Let \mathbf{U} be a set of views, and let \leq be a disclosure order for \mathbf{U} . Define $\mathcal{I} = \{(\Downarrow \mathbf{V}) : \mathbf{V} \subseteq \mathbf{U}\}$. Then (\mathcal{I}, \leq) is a lattice with details as follows:*

- (i) *LUB:* $(\Downarrow \mathbf{V}_1) \sqcup (\Downarrow \mathbf{V}_2) = \Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2)$.
- (ii) *GLB:* $(\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2) = \Downarrow (\mathbf{V}_1 \cap \mathbf{V}_2)$.
- (iii) *Top element* $\top = (\Downarrow \mathbf{U}) = \mathbf{U}$, *bottom element* $\perp = (\Downarrow \emptyset)$.

We call this lattice the *disclosure lattice over \mathbf{U}* . It is a strict generalization of the Lattice of Information [36].

Proof. For any pair of elements $(\Downarrow \mathbf{V}_1)$ and $(\Downarrow \mathbf{V}_2)$ in \mathcal{I} we know that $(\Downarrow \mathbf{V}_1) \leq (\Downarrow \mathbf{V}_2)$ if and only if $(\Downarrow (\Downarrow \mathbf{V}_1)) \subseteq (\Downarrow (\Downarrow \mathbf{V}_2))$ if and only if $(\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_2)$. Rather than showing that (\mathcal{I}, \leq) is a lattice, we prove the equivalent claim that (\mathcal{I}, \subseteq) is a lattice. We verify each of the properties of a lattice in turn:

- (i) **LUB:** Let \mathbf{V}_3 be such that $(\Downarrow \mathbf{V}_1) \subseteq (\Downarrow \mathbf{V}_3)$ and $(\Downarrow \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_3)$. Then $\mathbf{V}_1 \leq \mathbf{V}_3$ and $\mathbf{V}_2 \leq \mathbf{V}_3$, so that $(\mathbf{V}_1 \cup \mathbf{V}_2) \leq \mathbf{V}_3$. It follows that $\Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_3)$. In particular, by setting $(\Downarrow \mathbf{V}_3) = (\Downarrow \mathbf{V}_1) \sqcup (\Downarrow \mathbf{V}_2)$ we see that $\Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_1) \sqcup (\Downarrow \mathbf{V}_2)$.

On the other hand, $\mathbf{V}_1 \leq (\mathbf{V}_1 \cup \mathbf{V}_2)$ implies $(\Downarrow \mathbf{V}_1) \subseteq \Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2)$. Similarly, $(\Downarrow \mathbf{V}_2) \subseteq \Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2)$. It follows that $(\Downarrow \mathbf{V}_1) \sqcup (\Downarrow \mathbf{V}_2) \subseteq \Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2)$.

- (ii) **GLB:** Let \mathbf{V}_3 be such that $(\Downarrow \mathbf{V}_3) \subseteq (\Downarrow \mathbf{V}_1)$ and $(\Downarrow \mathbf{V}_3) \subseteq (\Downarrow \mathbf{V}_2)$. Then $(\Downarrow \mathbf{V}_3) \subseteq (\Downarrow \mathbf{V}_1 \cap \Downarrow \mathbf{V}_2)$. In particular, by setting $(\Downarrow \mathbf{V}_3) = (\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2)$ we see that $(\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_1 \cap \Downarrow \mathbf{V}_2)$.

On the other hand,

$$\Downarrow (\Downarrow \mathbf{V}_1 \cap \Downarrow \mathbf{V}_2) \subseteq \Downarrow (\Downarrow \mathbf{V}_1) = (\Downarrow \mathbf{V}_1)$$

$$\text{and } \Downarrow (\Downarrow \mathbf{V}_1 \cap \Downarrow \mathbf{V}_2) \subseteq \Downarrow (\Downarrow \mathbf{V}_2) = (\Downarrow \mathbf{V}_2)$$

$$\text{together imply } \Downarrow (\Downarrow \mathbf{V}_1 \cap \Downarrow \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2)$$

Combining this with the previous inequality, we find that

$$(\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_1 \cap \Downarrow \mathbf{V}_2) \subseteq \Downarrow (\Downarrow \mathbf{V}_1 \cap \Downarrow \mathbf{V}_2) \subseteq (\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2)$$

and hence, equality must hold.

(iii) **Top and bottom:**

To show that $\top = (\Downarrow \mathbf{U}) = \mathbf{U}$, we note that for any $\mathbf{V} \subseteq \mathbf{U}$ we have $\Downarrow \mathbf{V} \subseteq \mathbf{U}$. Since $\mathbf{U} \subseteq (\Downarrow \mathbf{U}) \subseteq \mathbf{U}$ we conclude that $\Downarrow \mathbf{U} = \mathbf{U}$ is the top element of our lattice.

To show that $\perp = (\Downarrow \emptyset)$, note that $\emptyset \subseteq \mathbf{V}$ for any $\mathbf{V} \subseteq \mathbf{U}$, so that $(\Downarrow \emptyset) \subseteq (\Downarrow \mathbf{V})$. \square

As an example, consider the `Meeting` relation from Figure 3.1 and suppose our universe \mathbf{U} consists of the four views in Figure 3.3(a). Letting \leq be the equivalent view rewriting ordering, the disclosure lattice for this \mathbf{U} is shown in Figure 3.3. The GLB of $\Downarrow \{V_2\}$ and $\Downarrow \{V_4\}$ is $\Downarrow \{V_5\}$. Their LUB is not $\Downarrow \{V_1\}$ but another properly lower element, accurately reflecting the fact that it is impossible to reconstitute the `Meeting` relation from the projections on its two attributes.

3.2.3 Disclosure Labelers

In this section we define disclosure labelers and explain under what conditions they exist. We begin with a set of security views \mathbf{S} , each of which reveals a known type of

information about the dataset. A labeler ℓ is a function that expresses the information revealed by an unknown set of queries \mathbf{Q} in terms of the information revealed by a subset $\mathbf{S}' \subseteq \mathbf{S}$.

It would seem that ℓ should map subsets of \mathbf{U} to subsets of \mathbf{S} . For technical reasons, we permit the labeler's output to range over elements of an arbitrary set \mathcal{F} , even if \mathcal{F} is not a power set. Furthermore, the output of ℓ does not need to be an element of \mathcal{F} so long as it is *equivalent* to an element of \mathcal{F} .

We place three additional restrictions on ℓ in order to ensure that the labels it finds are semantically meaningful. First, if $\mathbf{V} \in \mathcal{F}$ then $\ell(\mathbf{V})$ should reveal the same information as \mathbf{V} . This ensures that the labeler behaves correctly for “easy” inputs. It also means that the elements in \mathcal{F} are the *fixpoints* of ℓ , which motivates our choice of notation \mathcal{F} . Second, the labeler should never *underestimate* the amount of information disclosed by a set of queries. And third, ℓ should be monotonic – if one set of views reveals less than another then the label of the first set of views should be below the label of the second.

Definition 3.2.10. *Let \mathcal{F} be a subset of $\wp(\mathbf{U})$, the power set of \mathbf{U} , and assume \leq is a disclosure order. A **disclosure labeler** is a map $\ell : \wp(\mathbf{U}) \rightarrow \wp(\mathbf{U})$ such that*

- (i) *If $\mathbf{V}_1 \subseteq \mathbf{U}$ then $\ell(\mathbf{V}_1) \equiv \mathbf{V}_2$ for some $\mathbf{V}_2 \in \mathcal{F}$.*
- (ii) *If $\mathbf{V} \in \mathcal{F}$ then $\ell(\mathbf{V}) \equiv \mathbf{V}$.*
- (iii) *If $\mathbf{V} \subseteq \mathbf{U}$ then $\mathbf{V} \leq \ell(\mathbf{V})$.*
- (iv) *If $\mathbf{V}_1, \mathbf{V}_2 \subseteq \mathbf{U}$ and $\mathbf{V}_1 \leq \mathbf{V}_2$ then $\ell(\mathbf{V}_1) \leq \ell(\mathbf{V}_2)$.*

We call \mathcal{F} the set of *disclosure labels* for ℓ . The axioms defined above mirror those in the definition of an order-theoretic *closure operator* [25], defined as follows:

Definition 3.2.11. *Given a lattice (\mathcal{I}, \leq) , we say that a map $c : \mathcal{I} \rightarrow \mathcal{I}$ is a closure operator if for all $X, Y \in \mathcal{I}$,*

- (i) $X \leq c(X)$.
- (ii) $X \leq Y$ implies $c(X) \leq c(Y)$.
- (iii) $c(c(X)) = c(X)$.

The next Proposition makes the connection between disclosure labelers and closure operators explicit: every disclosure labeler induces a closure operator.

Proposition 3.2.12. *The operator c that maps every $X \in \mathcal{I}$ to $\Downarrow \ell(X)$ is a closure operator.*

Proof. Suppose $X, Y \in \mathcal{I}$. We verify each of the properties of a closure operator in turn:

- (i) $X \leq \ell(X) \equiv \Downarrow \ell(X) = c(X)$.
- (ii) Suppose $X \leq Y$. Then $c(X) = \Downarrow \ell(X) \equiv \ell(X) \leq \ell(Y) \equiv \Downarrow \ell(Y) = c(Y)$.
- (iii) The definition of a disclosure labeler implies that $c(X) = \Downarrow \ell(X) \equiv \ell(X) \equiv \mathbf{V}$ for some $\mathbf{V} \in \mathcal{F}$. We claim that $c(c(X)) \equiv \mathbf{V}$. The definition of a disclosure labeler implies that $\ell(\mathbf{V}) \equiv \mathbf{V}$. Since $\mathbf{V} \equiv c(X)$ implies $\mathbf{V} \leq c(X)$ we must have

$$c(X) \equiv \mathbf{V} \equiv \ell(\mathbf{V}) \leq \ell(c(X)) \equiv \Downarrow \ell(c(X)) = c(c(X))$$

An analogous argument in the other direction shows that $c(c(X)) \leq c(X)$, so that $c(X) \equiv c(c(X))$. It follows that $c(X) = c(c(X))$. □

Consider now the practical scenario where we start with a set of security views \mathbf{S} and would like to define a labeler where $\mathcal{F} = \wp(\mathbf{S})$, the powerset of \mathbf{S} . Unfortunately, we cannot always do this, as the following example shows.

Example 3.2.13. Consider V_2 and V_4 from Figure 3.3, and suppose we want to label queries with $\wp(\{V_2, V_4\})$. This gives the following $\mathcal{F} = \{\emptyset, \{V_2\}, \{V_4\}, \{V_2, V_4\}, \{V_1\}\}$.³

³The disclosure labeler axioms imply that \mathcal{F} contains the top element $\top = \Downarrow \{V_1\}$.

Suppose that, as in previous examples, \mathbf{U} consists of the four views in Figure 3.3. It turns out that there is no labeler for \mathbf{U} using \mathcal{F} as the set of disclosure labels. To see this, suppose a labeler does exist and consider what $\ell(V_5)$ might be. Since $\{V_5\} \leq \{V_2\}$, we know that $\ell(\{V_5\}) \leq \ell(\{V_2\}) \equiv \{V_2\}$. Similarly, since $\{V_5\} \leq \{V_4\}$, we know that $\ell(\{V_5\}) \leq \ell(\{V_4\}) \equiv \{V_4\}$. Since $\ell(\{V_5\}) \in \mathcal{F}$, we are forced to conclude that $\ell(\{V_5\}) = \emptyset$. However, $\{V_5\} \not\leq \emptyset$, violating condition (iii) of Definition 4.3.4.

The conditions in Definition 4.3.4 are much stronger than they first appear. It is not always possible to define a disclosure labeler for a given \mathcal{F} ; however, when a disclosure labeler *does* exist, it is unique up to equivalence. To formalize and prove this, it is necessary to understand how labelers interact with the disclosure lattice defined in Section 3.2.2.

If we apply a disclosure labeler ℓ to each element of \mathcal{I} (the disclosure lattice of \mathbf{U}), we obtain a new lattice $\ell(\mathcal{I})$. We call this lattice the *lattice of disclosure labels*. Roughly speaking, each element in this lattice corresponds to the information revealed by an element of \mathcal{F} . More formally, it is possible to show that $\ell(\mathcal{I}) = \{\Downarrow \mathbf{V} : \mathbf{V} \in \mathcal{F}\}$. For arbitrary $\mathbf{V}_1, \mathbf{V}_2 \in \mathcal{F}$ this lattice is guaranteed to contain the GLB $(\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2)$. It will also contain a suitable $\Downarrow \mathbf{V}_3$, for $\mathbf{V}_3 \in \mathcal{F}$, which can serve as a LUB. However, this new LUB may be higher in the original disclosure lattice than $(\Downarrow \mathbf{V}_1) \sqcup (\Downarrow \mathbf{V}_2)$.

Theorem 3.2.14 (Labeling on the disclosure lattice). *Define $\ell(\mathcal{I}) = \{\Downarrow \ell(\mathbf{V}) : \mathbf{V} \in \mathcal{I}\}$. Then $(\ell(\mathcal{I}), \leq)$ is a lattice, with GLB and LUB as follows.*

- (i) *GLB:* $(\Downarrow \mathbf{V}_1) \sqcap_\ell (\Downarrow \mathbf{V}_2) = (\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2)$.
- (ii) *LUB:* $(\Downarrow \mathbf{V}_1) \sqcup_\ell (\Downarrow \mathbf{V}_2) = \Downarrow \ell((\Downarrow \mathbf{V}_1) \sqcup (\Downarrow \mathbf{V}_2))$.

We begin by proving the following Lemma.

Lemma 3.2.15. *If ℓ is a disclosure labeler and $\mathbf{V}_1 \leq \ell(\mathbf{V}_2)$ then $\ell(\mathbf{V}_1) \leq \ell(\mathbf{V}_2)$.*

Proof. $\mathbf{V}_1 \leq \ell(\mathbf{V}_1) \leq \ell(\ell(\mathbf{V}_2)) \equiv \ell(\mathbf{V}_2)$. □

We are now ready to prove the Theorem 3.2.14.

Proof of Theorem. We handle each part of the theorem separately:

(i) Let $\mathbf{V}_1, \mathbf{V}_2 \in \mathcal{F}$, and let \mathbf{V}_3 be such that $(\Downarrow \mathbf{V}_3) = (\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2)$. We first note that $\ell(\Downarrow \mathbf{V}_3) \leq \ell(\Downarrow \mathbf{V}_i) \equiv (\Downarrow \mathbf{V}_i)$ for $i = 1, 2$, and therefore $\ell(\Downarrow \mathbf{V}_3) \leq (\Downarrow \mathbf{V}_3)$. On the other hand, $(\Downarrow \mathbf{V}_3) \leq \ell(\Downarrow \mathbf{V}_3)$. It follows that $(\Downarrow \mathbf{V}_3) \equiv \ell(\Downarrow \mathbf{V}_3)$, and therefore $\Downarrow \mathbf{V}_3 = \Downarrow \ell(\Downarrow \mathbf{V}_3)$.

(ii) Let \mathbf{V}_3 be such that $(\Downarrow \mathbf{V}_3) = (\Downarrow \mathbf{V}_1) \sqcup (\Downarrow \mathbf{V}_2)$. Since $(\Downarrow \mathbf{V}_1) \leq (\Downarrow \mathbf{V}_3)$ we know that $\Downarrow \mathbf{V}_1 = \Downarrow \ell(\mathbf{V}_1) \leq \Downarrow \ell(\mathbf{V}_3)$. Similarly, $\Downarrow \mathbf{V}_2 = \Downarrow \ell(\mathbf{V}_2) \leq \Downarrow \ell(\mathbf{V}_3)$. It follows that $(\Downarrow \mathbf{V}_1 \sqcup_\ell \Downarrow \mathbf{V}_2) \leq \Downarrow \ell(\Downarrow \mathbf{V}_3)$.

On the other hand, $(\Downarrow \mathbf{V}_i) \leq (\Downarrow \mathbf{V}_1 \sqcup_\ell \Downarrow \mathbf{V}_2)$ for each $i = 1, 2$, so that $(\Downarrow \mathbf{V}_3) \leq (\Downarrow \mathbf{V}_1 \sqcup_\ell \Downarrow \mathbf{V}_2)$. It follows that

$$\Downarrow \ell(\Downarrow \mathbf{V}_3) \leq \Downarrow \ell(\Downarrow \mathbf{V}_1 \sqcup_\ell \Downarrow \mathbf{V}_2) = (\Downarrow \mathbf{V}_1 \sqcup_\ell \Downarrow \mathbf{V}_2)$$

and therefore $\Downarrow \ell(\Downarrow \mathbf{V}_3) = (\Downarrow \mathbf{V}_1 \sqcup_\ell \Downarrow \mathbf{V}_2)$, as desired. □

Theorem 3.2.14 provides the insight to characterize when a set \mathcal{F} can be used to formulate a disclosure labeler. Existence is characterized using *closure systems* from lattice theory.

Definition 3.2.16. We say that $S \subseteq \mathcal{I}$ is a **closure system** for a finite lattice (\mathcal{I}, \leq) if

- (i) For each $X_1, X_2 \in S$, we have $X_1 \sqcap X_2 \in S$, and
- (ii) S contains the (unique) maximal element of \mathcal{I} .

The following Theorem shows that there is a bijective correspondence between disclosure labelers and closure systems. A disclosure labeler is uniquely determined by its fixpoints, and the fixpoints must form a closure system.

Theorem 3.2.17 (Labeler Existence). *Let $\mathcal{F} \subseteq \wp(\mathbf{U})$, and assume \leq is a disclosure order. Then $\mathcal{I}_{\mathcal{F}} = \{\Downarrow \mathbf{V} : \mathbf{V} \in \mathcal{F}\}$ is a closure system if and only if there exists a disclosure labeler ℓ with domain \mathbf{U} and image \mathcal{F} .*

Proof. Theorem 3.2.9 implies that if ℓ is a disclosure labeler for $(\mathbf{U}, \mathcal{F}, \leq)$ then $\{\Downarrow \mathbf{V} : \mathbf{V} \in \mathcal{F}\}$ is closed under greatest lower bounds. Since \mathbf{U} is a maximal element of (\mathcal{I}, \leq) , $\ell(\mathbf{U}) \equiv \mathbf{U}$, and therefore $(\Downarrow \mathbf{U}) \in \mathcal{S}$. It follows that $(\Downarrow \mathbf{U})$, the (unique) maximal element of \mathcal{I} , is contained in \mathcal{S} , so that $\{\Downarrow \mathbf{V} : \mathbf{V} \in \mathcal{F}\}$ is a closure system.

For the other direction, suppose $\mathcal{I}_{\mathcal{F}} = \{\Downarrow \mathbf{V} : \mathbf{V} \in \mathcal{F}\}$ is a closure system. Define

$$\ell(\mathbf{V}) = \bigsqcap \{\Downarrow \mathbf{V}' \in \mathcal{F} : \mathbf{V} \leq \mathbf{V}'\}$$

We claim that ℓ is a disclosure labeler. Since $\mathcal{I}_{\mathcal{F}}$ induces a closure system, $\Downarrow \ell(\mathbf{V}) \in \mathcal{I}_{\mathcal{F}}$ for all possible \mathbf{V} . Furthermore, if $\mathbf{V} \in \mathcal{F}$ then by construction, $\mathbf{V} \equiv \ell(\mathbf{V})$. Next, $\ell(\mathbf{V})$ is a GLB of elements \mathbf{V}' such that $\mathbf{V} \leq \mathbf{V}'$, and therefore $\mathbf{V} \leq \ell(\mathbf{V})$. If $\mathbf{V}_1 \leq \mathbf{V}_2$ then

$$\ell(\mathbf{V}_1) = \bigsqcap \{\Downarrow \mathbf{V}' \in \mathcal{F} : \mathbf{V}_1 \leq \mathbf{V}'\} \leq \bigsqcap \{\Downarrow \mathbf{V}' \in \mathcal{F} : \mathbf{V}_2 \leq \mathbf{V}'\} = \ell(\mathbf{V}_2)$$

and therefore $\ell(\mathbf{V}_1) \leq \ell(\mathbf{V}_2)$. Hence, ℓ is a disclosure labeler for \mathcal{F} , as desired. \square

Intuitively, \mathcal{F} can be used to formulate a disclosure labeler if the corresponding set it induces in the disclosure lattice is both closed under GLB and contains \top . If a labeler does exist, it is unique up to equivalence:

Proposition 3.2.18. *If ℓ_1 and ℓ_2 are both disclosure labelers for $(\mathbf{U}, \mathcal{F}, \leq)$ then $\ell_1(\mathbf{V}) \equiv \ell_2(\mathbf{V})$ for all $\mathbf{V} \subseteq \mathbf{U}$.*

Proof. For any $\mathbf{V} \subseteq \mathbf{U}$ we have $\mathbf{V} \leq \ell_2(\mathbf{V})$, so that $\ell_1(\mathbf{V}) \leq \ell_2(\mathbf{V})$ by Lemma 3.2.15. Similarly, $\mathbf{V} \leq \ell_1(\mathbf{V})$, so that $\ell_2(\mathbf{V}) \leq \ell_1(\mathbf{V})$ by Lemma 3.2.15. It follows that $\ell_1(\mathbf{V}) \leq \ell_2(\mathbf{V}) \leq \ell_1(\mathbf{V})$, and therefore $\ell_1(\mathbf{V}) \equiv \ell_2(\mathbf{V})$. \square

This allows us to formulate the following definition:

Definition 3.2.19 (Inducing labelers). *Let $\mathcal{F} \subseteq \wp(\mathbf{U})$. We say \mathcal{F} induces a disclosure labeler on \mathbf{U} if it satisfies the condition in Theorem 3.2.17. We call the labeler ℓ from Theorem 3.2.17 the labeler induced by \mathcal{F} .*

If \mathcal{F} induces a disclosure labeler, the following algorithm is a naïve but correct implementation of that labeler. It assumes that no two distinct elements of \mathcal{F} are equivalent. The algorithm sorts the elements of \mathcal{F} in order of increasing disclosure (Lines 2–3) and finds the first element in the new order that reveals at least as much information as \mathbf{V} (Lines 4–8).

```

1: procedure NAÏVELABEL( $\mathcal{F}, \mathbf{V}$ )
2:   Let  $\mathcal{F}[1], \dots, \mathcal{F}[n]$  be the elements of  $\mathcal{F}$ .
3:   Sort  $\mathcal{F}$  so that if  $\mathcal{F}[i] \leq \mathcal{F}[j]$  then  $i \leq j$ .
4:   for  $i \leftarrow 1, 2, \dots, n$  do
5:     if  $\mathbf{V} \leq \mathcal{F}[i]$  then
6:       return  $\mathcal{F}[i]$ 
7:     end if
8:   end for
9:   return  $\top$ 
10: end procedure

```

We now verify that NAÏVELABEL is in fact a disclosure labeler.

Proposition 3.2.20. *If \mathcal{F} induces a closure system then NAïVELABEL is a disclosure labeler for $(\mathbf{U}, \mathcal{F}, \leq)$.*

Proof. We verify each of the disclosure labeler axioms in turn. By construction, every element in $\wp(\mathbf{U})$ is mapped to an element of \mathcal{F} . The check on line 5 ensures that $\text{NAïVELABEL}(\mathcal{F}, \mathbf{V}) \leq \mathbf{V}$. In the special case where $\mathbf{V} \in \mathcal{F}$, we must show that $\text{NAïVELABEL}(\mathcal{F}, \mathbf{V}) \equiv \mathbf{V}$; this is true because the algorithm performs a topological sort on the elements of \mathcal{F} (Line 3) before iterating through them. Finally, we must show that if $\mathbf{V}_1 \leq \mathbf{V}_2$ then $\text{NAïVELABEL}(\mathcal{F}, \mathbf{V}_1) \leq \text{NAïVELABEL}(\mathcal{F}, \mathbf{V}_2)$; this is true because \leq is transitive and the elements of \mathcal{F} are checked in topological order (Lines 4-8). \square

This completes our presentation of disclosure labelers. Note that the output of a labeler satisfies our desiderata: It is a data-derived measure of disclosure because it is a mathematical function of the data needed to answer the queries. The output is also semantically meaningful, as it expresses disclosure by relating it to security views that the user understands, and the model is expressive due to the ability to choose a large and complex \mathcal{F} .

3.2.4 From Labelers to Security Policies

We have defined *disclosure labelers* which allow us to restate the information revealed by an unknown set of queries in terms of the information revealed by a much smaller, known, set of security views. Labelers are useful because they allow the formulation of semantically meaningful security policies; we explain this now in more detail.

As we saw, the information associated with the disclosure labels for a given labeler can be represented as a lattice $\ell(\mathcal{I})$. Conceptually, a security policy is a cut in this lattice:

a set of queries whose label is below the cut can be answered, but a set of queries whose label falls above the cut cannot. We can formally represent a security policy as the set of elements in the lattice that are below the desired cut.

Definition 3.2.21 (Security policy). *A security policy \mathcal{P} for labeler ℓ is a subset of the elements in the lattice of disclosure labels for ℓ .*

For example, let \mathbf{U} contain the four views in Figure 3.3, and let ℓ be a trivial disclosure labeler that maps every subset of \mathbf{U} to itself. $\mathcal{P} = \{\perp, \Downarrow \{V_5\}, \Downarrow \{V_2\}, \Downarrow \{V_4\}\}$ represents a policy in which either the first or the second attribute of `Meeting` may be disclosed, but not both. This is an example of a *Chinese Wall* policy [17] and shows that our framework is powerful enough to express fairly complex policies cleanly.

An important restriction is that policies must be *internally consistent* in the sense that if $\mathbf{V} \leq \mathbf{V}'$ and $\Downarrow \mathbf{V}' \in \mathcal{P}$ then $\Downarrow \mathbf{V} \in \mathcal{P}$. In our running example, a principal who can view the entirety of the `Meeting` relation should also be permitted to view the projections on each attribute.

In practice, we assume queries arrive in the system one at a time. A reference monitor is an algorithm that inspects each query and accepts or rejects it to ensure the policy is never violated. A simple algorithm for enforcing a security policy \mathcal{P} while answering a set of queries \mathbf{Q} is given below.

```

1:  $\mathcal{L}_{total} \leftarrow \emptyset$ 
2: for  $Q \in \mathbf{Q}$  do
3:    $\mathcal{L}_{new} \leftarrow \ell(Q \cup \mathcal{L}_{total})$ 
4:   if  $(\Downarrow \mathcal{L}_{new}) \in \mathcal{P}$  then
5:     answer  $Q$ 
6:      $\mathcal{L}_{total} \leftarrow \mathcal{L}_{new}$ 
7:   else

```

$$\begin{array}{ll}
V_3(x, y, z) :- C(x, y, z) & V_9(x) :- C(x, y, z) \\
V_6(x, y) :- C(x, y, z) & V_{10}(y) :- C(x, y, z) \\
V_7(x, z) :- C(x, y, z) & V_{11}(z) :- C(x, y, z) \\
V_8(y, z) :- C(x, y, z) & V_{12}() :- C(x, y, z)
\end{array}$$

Figure 3.4: All relational projections of **Contact**.

```

8:     refuse  $Q$ 
9:     end if
10: end for

```

The algorithm processes incoming queries in \mathbf{Q} one at a time (Line 2). It first computes the total information disclosed if the query would be answered (Line 3). If such disclosure is permitted by the policy (Line 4), the query is answered and the cumulative disclosure \mathcal{L}_{total} is updated (Lines 5 and 6). This completes our presentation of disclosure labelers. We now show how they can be made practical.

3.3 Generating labelers

The naïve disclosure labeling algorithm proposed in Section 3.2.3 runs in time that is linear in the size of the set \mathcal{F} . Unfortunately, \mathcal{F} can easily become very large, since it generally contains all possible subsets of a set of security views \mathbf{S} . In fact, as the next example demonstrates, even \mathbf{S} itself can grow quite large.

Example 3.3.1. Consider a generalization of the example used in Figure 3.3. Suppose we have an n -attribute relation R and wish to label each query over R with the set of relational projections on R that is required to answer it. There are 2^n possible projections on R – Figure 3.4 shows all of them for the three-attribute relation **Contact** from Figure

3.1, where the relation name is abbreviated as C . The set \mathcal{F} would need to account for all possible subsets of these projections, for a total size that is doubly-exponential in n .

It is clearly impractical to work with such a large \mathcal{F} . Fortunately, we do not need to represent all of \mathcal{F} explicitly. We can instead work with a smaller subset of \mathcal{F} and in some sense materialize any remaining elements as they are needed. This section focuses on the problem of finding a suitable subset of \mathcal{F} which is as small as possible.

We assume the existence of two black-box algorithms that depend on \leq and on \mathbf{U} . The first takes as input subsets \mathbf{V}_1 and \mathbf{V}_2 of \mathbf{U} , and determines whether $\mathbf{V}_1 \leq \mathbf{V}_2$ in the disclosure order. The second, written $GLB(\mathbf{V}_1, \mathbf{V}_2)$, finds a set of views \mathbf{V}_3 such that $(\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2) = (\Downarrow \mathbf{V}_3)$. GLB generalizes to handle an arbitrarily large number of input arguments in the obvious way. Section 3.4 contains a concrete instantiation of these algorithms for the case of equivalent view rewriting on conjunctive queries under set semantics.

3.3.1 Downward Generating Sets

Suppose we have a set \mathcal{F} that induces a labeler. This means it is closed under the GLB operation. Therefore, some elements of \mathcal{F} are “redundant” in the sense that they can be computed by taking $GLBs$ of other elements. Such redundant elements can be removed to yield a smaller \mathcal{F}_d that can replace \mathcal{F} for practical purposes.

Definition 3.3.2 (Downward generating set). *Given a set \mathcal{F} , we call $\mathcal{F}_d \subseteq \mathcal{F}$ a downward generating set for \mathcal{F} if for every $\mathbf{V} \in \mathcal{F}$ there exist $\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_n \in \mathcal{F}_d$ such that $\mathbf{V} \equiv GLB(\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_n)$.*

Every \mathcal{F} that induces a labeler has a unique minimal downward generating set.

Theorem 3.3.3. *If \mathcal{F} induces a labeler then there exists a downward-generating set \mathcal{F}_d for \mathcal{F} which is minimal under the usual set ordering. The elements of \mathcal{F}_d are uniquely determined up to equivalence.*

Proof. Observe that \mathcal{F} must have at least one downward generating set: itself. Since \mathcal{F} is finite by assumption, it must have a downward generating subset \mathcal{F}_d of minimal size; this subset must also be minimal under set semantics. It remains for us to show that \mathcal{F}_d is unique up to equivalence.

Let us suppose that \mathcal{F}_d and \mathcal{F}'_d are both downward generating subsets of \mathcal{F} that are minimal under set semantics. We will show that each $\mathbf{V} \in \mathcal{F}_d$ is equivalent to an element $\mathbf{V}' \in \mathcal{F}'_d$. Since $\mathcal{F}_d \subseteq \mathcal{F}$, each $\mathbf{V} \in \mathcal{F}_d$ is equivalent to $(\mathbf{V}'_1 \sqcap \mathbf{V}'_2 \sqcap \dots \sqcap \mathbf{V}'_m)$, where $\mathbf{V}'_1, \dots, \mathbf{V}'_m \in \mathcal{F}'_d$. Similarly, each $\mathbf{V}'_i \in \mathcal{F}'_d$ can be expressed as a greatest lower bound $(\mathbf{V}_{i,1} \sqcap \mathbf{V}_{i,2} \sqcap \dots \sqcap \mathbf{V}_{i,n_i})$ of elements in \mathcal{F}_d . Writing this out, we see that

$$\mathbf{V} = (\mathbf{V}_{1,1} \sqcap \dots \sqcap \mathbf{V}_{1,n_1}) \sqcap \dots \sqcap (\mathbf{V}_{m,1} \sqcap \dots \sqcap \mathbf{V}_{m,n_m})$$

By construction, $\mathbf{V} \leq \mathbf{V}'_i$ and $\mathbf{V}'_i \leq \mathbf{V}_{i,j}$ for each $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n_i$. There are now two cases. In the first case, \mathbf{V} is equivalent to some \mathbf{V}'_i , in which case we're done. In the second case, \mathbf{V} is *not* equal to any \mathbf{V}'_i . In the latter case, none of the $\mathbf{V}_{i,j}$ can be equal to \mathbf{V} . However, in this case, \mathcal{F}_d cannot be minimal because the set difference $\mathcal{F}_d - \{\mathbf{V}\}$ is GLB-dense in \mathcal{F} . Hence, we have a contradiction. \square

Given \mathcal{F} , a minimal downward generating set can be computed by iteratively removing elements of \mathcal{F} that are equivalent to the *GLB* of a subset of the elements still left.

Example 3.3.4. Continuing with Example 3.3.1, let \leq be the equivalent view rewriting ordering. A downward generating set for the original \mathcal{F} is $\emptyset(\{V_3, V_6, V_7, V_8\})$. The reason

for this will become clearer in Section 3.4 when we explain how GLB is computed for the equivalent view rewriting order. It turns out that

$$GLB(\{V_6\}, \{V_7\}) \equiv \{V_9\}$$

$$GLB(\{V_6\}, \{V_8\}) \equiv \{V_{10}\}$$

$$GLB(\{V_7\}, \{V_8\}) \equiv \{V_{11}\}$$

$$GLB(\{V_6\}, \{V_7\}, \{V_8\}) \equiv \{V_{12}\}$$

This should provide intuition as to why we removed the last four views in Figure 3.4. Note that the size of \mathcal{F}_d is still exponential in the number of attributes of `Contact`.

Given a downward generating set \mathcal{F}_d , the procedure $GLBLABEL$ defined below shows how to use it for query labeling. The algorithm iterates over all elements of \mathcal{F}_d (Line 3) and computes a running GLB of those elements that disclose at least as much information as \mathbf{V} (Lines 4-6).

```

1: procedure  $GLBLABEL(\mathcal{F}_d, \mathbf{V})$ 
2:    $L \leftarrow \top$ 
3:   for  $\mathbf{V}' \in \mathcal{F}_d$  do
4:     if  $\mathbf{V} \preceq \mathbf{V}'$  then
5:        $L \leftarrow GLB(L, \mathbf{V}')$ 
6:     end if
7:   end for
8:   return  $L$ 
9: end procedure

```

We next verify the correctness of this updated procedure.

Theorem 3.3.5. *If \mathcal{F} induces a closure system and \mathcal{F}_d is GLB -dense in \mathcal{F} then $GLBLABEL$ is a disclosure labeler for $(\mathbf{U}, \mathcal{F}, \preceq)$.*

Proof. We verify each of the disclosure labeler properties in turn:

- (i) If $\mathbf{V}_1 \subseteq \mathbf{U}$ then $\text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}_1) \equiv \mathbf{V}_2$ for some $\mathbf{V}_2 \in \mathcal{F}$.

The output of the algorithm above is equal to the GLB of a subset of the elements of \mathcal{F}_d . By assumption, every element of \mathcal{F}_d is equivalent to an element of \mathcal{F} . Since \mathcal{F} is a closure system, we can show that the GLB of any subset of the elements of \mathcal{F} is equivalent to another element of \mathcal{F} by induction on the size of the subset, say n . When $n = 0$, the output is \top , which is equivalent to an element of \mathcal{F} by definition. When $n = 1$, we map an element of \mathcal{F} to itself. When $n > 1$,

$$X_1 \sqcap \dots \sqcap X_{n-1} \sqcap X_n = (X_1 \sqcap \dots \sqcap X_{n-1}) \sqcap X_n$$

where $(X_1 \sqcap \dots \sqcap X_{n-1})$ and X_n are equivalent to elements of \mathcal{F} , and therefore $(X_1 \sqcap \dots \sqcap X_{n-1} \sqcap X_n)$ is also equivalent to an element of \mathcal{F} .

- (ii) If $\mathbf{V} \in \mathcal{F}$ then $\text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}) \equiv \mathbf{V}$.

Since \mathcal{F}_d is GLB-dense in \mathcal{F} , there is a subset \mathbf{V}' of the elements of \mathcal{F}_d whose GLB is equivalent to \mathbf{V} . Every element of \mathbf{V}' is above \mathbf{V} in the disclosure lattice, and therefore will be included in the GLB computed by the algorithm above. It follows that $\text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}) \leq \mathbf{V}$. On the other hand, every element in the GLB computed by GLBLABEL is above \mathbf{V} in the disclosure lattice, and therefore $\mathbf{V} \leq \text{GLBLABEL}(\mathcal{F}_d, \mathbf{V})$. Equivalence follows.

- (iii) If $\mathbf{V} \subseteq \mathbf{U}$ then $\mathbf{V} \leq \text{GLBLABEL}(\mathcal{F}_d, \mathbf{V})$.

Every element in the GLB computed by GLBLABEL is above \mathbf{V} in the disclosure lattice, and therefore $\mathbf{V} \leq \text{GLBLABEL}(\mathcal{F}_d, \mathbf{V})$. If there are no elements in \mathbf{V} that are above \mathbf{V} then the labeler's output will be \top , which is above \mathbf{V} by definition.

- (iv) If $\mathbf{V}_1, \mathbf{V}_2 \subseteq \mathbf{U}$ and $\mathbf{V}_1 \leq \mathbf{V}_2$ then $\text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}_1) \leq \text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}_2)$.

If $\mathbf{V}_1 \leq \mathbf{V}_2$ then every element of \mathcal{F}_d that is above \mathbf{V}_2 in the disclosure lattice will also be above \mathbf{V}_1 . This means that the elements whose GLB is returned as the output of $\text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}_2)$ form a subset of those whose GLB is returned as the output of $\text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}_1)$, and therefore

$$\text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}_1) \leq \text{GLBLABEL}(\mathcal{F}_d, \mathbf{V}_2)$$

as desired. □

Last but not least, downward generating sets obviate the problem of checking whether a given collection of sets of security views induces a labeler. It turns out that we can extend *any* set \mathcal{G} to one that induces a labeler by closing it under the *GLB* operation.

Theorem 3.3.6. *If \mathcal{G} is a collection of sets of views that contains the top element ($\Downarrow \mathbf{U}$) then there is a set $\mathcal{F} \supseteq \mathcal{G}$ such that (i) \mathcal{F} induces a disclosure labeler and (ii) \mathcal{G} is a downward generating set for \mathcal{F} . The elements of \mathcal{F} are unique up to equivalence.*

Proof. We let \mathcal{F} be a minimal collection of sets such that

- (i) $\mathcal{F} \supset \mathcal{G}$,
- (ii) $\mathbf{U} \in \mathcal{F}$, and
- (iii) If $\mathbf{V}_1, \mathbf{V}_2 \in \mathcal{F}$ then there is some $\mathbf{V}_3 \in \mathcal{F}$ such that $(\Downarrow \mathbf{V}_1) \sqcap (\Downarrow \mathbf{V}_2) = (\Downarrow \mathbf{V}_3)$.

Then $\mathcal{I}_{\mathcal{F}} = \{\Downarrow \mathbf{V} : \mathbf{V} \in \mathcal{F}\}$ is a closure system and every element of \mathcal{F} is equivalent to a GLB of zero or more elements of \mathcal{G} . Hence \mathcal{G} is a downward generating set for \mathcal{F} .

Now suppose that there is some other closure system \mathcal{F}' that satisfies the required conditions. Any $\mathbf{V}' \in \mathcal{F}'$ can be expressed as a GLB of elements in \mathcal{F} , so that \mathbf{V}' is equivalent to an element of \mathcal{F} . It follows that $\mathcal{I}_{\mathcal{F}'} \subseteq \mathcal{I}_{\mathcal{F}}$. An analogous argument shows that $\mathcal{I}_{\mathcal{F}} \subseteq \mathcal{I}_{\mathcal{F}'}$, and therefore $\mathcal{I}_{\mathcal{F}} = \mathcal{I}_{\mathcal{F}'}$. □

This is in some sense the converse of Theorem 3.3.3 and in practice removes the need for checking whether \mathcal{G} induces a labeler. If it does not, we know we can extend it to one that does. In fact, we are free to work directly with \mathcal{G} since it is a downward generating set for that labeler.

3.3.2 Generating Sets

Although the reduction in size from \mathcal{F} to \mathcal{F}_d is substantial, Example 3.3.4 demonstrates that the size of \mathcal{F}_d can still be exponential in the number of attributes in the database schema. The question arises whether we can find a smaller subset of \mathcal{F}_d and use that for labeling instead. The answer is *yes*; however, we must place two key restrictions on \mathcal{F} and \mathbf{U} . First, \mathcal{F} must be *precise* in the following sense:

Definition 3.3.7 (Precise labeler). *Suppose \mathcal{F} contains \emptyset , and additionally if $(\Downarrow \mathbf{V}_1) \in \mathcal{F}$ and $(\Downarrow \mathbf{V}_2) \in \mathcal{F}$ then $\Downarrow (\mathbf{V}_1 \cup \mathbf{V}_2) \in \mathcal{F}$. We say \mathcal{F} induces a precise labeler.*

This definition states that \mathcal{F} – or rather, its extension to the disclosure lattice on \mathbf{U} – is closed under the lattice LUB operator. The intuition for the term *precise* is the following. Return to Figure 3.3 and suppose $\mathcal{F} = \{\emptyset, \{V_5\}, \{V_2\}, \{V_4\}, \top\}$, \mathbf{U} is the set of all conjunctive queries over `Meeting` and \leq is the equivalent view rewriting order. \mathcal{F} induces a labeler ℓ over \mathbf{U} , but it is not precise. Specifically, $\ell(\{V_2, V_4\}) = \top$, which is properly higher in the disclosure order than $\{V_2, V_4\}$, so the labeler exhibits some imprecision on this set of views.

The second restriction we need relates to the universe \mathbf{U} under the ordering \leq . Intuitively, we want to ensure that if the answers \mathbf{V}_1 are determined by \mathbf{V}_2 and \mathbf{V}_3 jointly then they must either be determined by \mathbf{V}_2 alone or by \mathbf{V}_3 alone.

Definition 3.3.8 (Decomposability). *We say a set \mathbf{U} is decomposable under \leq if for every $\mathbf{V}_1, \mathbf{V}_2 \subseteq \mathbf{U}$ and every $\{V\} \leq \mathbf{V}_1 \cup \mathbf{V}_2$ we have either $\{V\} \leq \mathbf{V}_1$ or $\{V\} \leq \mathbf{V}_2$.*

The following Theorem is an immediate consequence of Theorem 4 of Landauer’s 1993 paper *A Lattice of Information* [36].

Theorem 3.3.9. *If \mathbf{U} is decomposable under \leq then the corresponding disclosure lattice (\mathcal{I}, \leq) is distributive.*

Assume now that \mathbf{U} is decomposable under \leq and \mathcal{F} induces a precise labeler. We can define the concept of a (full) *generating set* \mathcal{F}_{gen} for \mathcal{F} . As before, a set \mathcal{F}_{gen} generates \mathcal{F} if \mathcal{F} can be obtained by closing \mathcal{F}_{gen} under both GLBs and LUBs. If \mathbf{U} is decomposable then it suffices to ensure that every element of \mathcal{F} is a LUB of GLBs of elements of \mathcal{F}_{gen} ; this fact follows immediately from distributivity.

Definition 3.3.10 (Generating set). *We say that \mathcal{F}_{gen} is a **generating set** for \mathcal{F} if every element of \mathcal{F} is equivalent to the union of GLBs of elements of \mathcal{F}_{gen} .*

Given a generating set \mathcal{F}_{gen} , we can use it for query labeling. The following algorithm processes \mathbf{V} one view at a time (Line 3) and computes a running union of the labels for the views (Line 4).

```

1: procedure GENLABEL( $\mathcal{F}_{gen}, \mathbf{V}$ )
2:   result  $\leftarrow \emptyset$ 
3:   for each  $V \in \mathbf{V}$  do
4:     result  $\leftarrow$  result  $\cup$  GLBLABEL( $\mathcal{F}_{gen}, \{V\}$ )
5:   end for
6:   return result
7: end procedure

```

We now verify the correctness of this procedure.

Theorem 3.3.11. *Suppose that (i) \mathcal{F} induces a precise closure system, (ii) \mathcal{F}_{gen} is a generating subset of \mathcal{F} , and (iii) \mathbf{U} is decomposable. Then GENLABEL is a disclosure labeler for $(\mathbf{U}, \mathcal{F}, \leq)$.*

Proof. Fix a set \mathbf{V} , and let V_1, V_2, \dots, V_n be the distinct elements of \mathbf{V} . Then

$$\text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}) = \bigcup_{i=1}^n \text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\})$$

We verify each of the disclosure labeler axioms in turn:

(i) If $\mathbf{V} \subseteq \mathbf{U}$ then $\text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}) \equiv \mathbf{V}'$ for some $\mathbf{V}' \in \mathcal{F}$.

The proof of correctness for GLBLABEL shows that each $\text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\})$ is equivalent to a GLB of elements of \mathcal{F}_{gen} . Since \mathcal{F}_{gen} is a generating set for \mathcal{F} , we conclude that $\text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V})$ is equivalent to an element of \mathcal{F} .

(ii) If $\mathbf{V} \in \mathcal{F}$ then $\text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}) \equiv \mathbf{V}$.

We will show that $\mathbf{V} \leq \text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V})$ and $\text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}) \leq \mathbf{V}$. The first direction is covered by part (iii) below; we still need to verify the second direction.

Recall that

$$\text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}) = \bigcup_{i=1}^n \text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\})$$

It therefore suffices to show that $\text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\}) \leq \mathbf{V}$ for each $V_i \in \mathbf{V}$. Fix $V_i \in \mathbf{V}$. Since \mathbf{U} is decomposable and $\{V_i\} \leq \mathbf{V}$, there must be some $\mathbf{V}' \in \mathcal{F}$ such that $\{V_i\} \leq \mathbf{V}' \leq \mathbf{V}$ and \mathbf{V}' is equivalent to a GLB of elements in \mathcal{F}_{gen} , say $F_{i,1}, F_{i,2}, \dots, F_{i,m_i}$. Now $\{V_i\} \leq \{F_{i,j}\}$ for each $j = 1, 2, \dots, m_i$, so that

$$\text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\}) \leq \text{GLBLABEL}(\mathcal{F}_{gen}, \{F_{i,j}\}) \equiv \{F_{i,j}\}$$

for each $j = 1, 2, \dots, m_i$ and therefore

$$\text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\}) \leq \prod_{j=1}^{m_i} \text{GLBLABEL}(\mathcal{F}_{gen}, \{F_{i,j}\}) \equiv \prod_{j=1}^{m_i} \{F_{i,j}\} \equiv \mathbf{V}' \leq \mathbf{V}$$

(iii) If $\mathbf{V} \subseteq \mathbf{U}$ then $\mathbf{V} \leq \text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V})$.

For $i = 1, 2, \dots, n$ we have

$$\{V_i\} \leq \text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\}) \leq \text{GLBLABEL}(\mathcal{F}_{gen}, \mathbf{V})$$

and therefore

$$\mathbf{V} = \bigcup_{i=1}^n \{V_i\} \leq \bigcup_{i=1}^n \text{GLBLABEL}(\mathcal{F}_{gen}, \mathbf{V}) = \text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V})$$

(iv) If $\mathbf{V}, \mathbf{V}' \subseteq \mathbf{U}$ and $\mathbf{V} \leq \mathbf{V}'$ then $\text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}) \leq \text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}')$.

For each $V_i \in \mathbf{V}$ there exists $V'_i \in \mathbf{V}'$ such that $\{V_i\} \leq \{V'_i\}$ because \mathcal{F} is decomposable. It follows that $\{V_i\} \leq \{V'_i\} \leq \text{GLBLABEL}(\mathcal{F}_{gen}, \{V'_i\}) \leq \text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}')$ for each $i = 1, 2, \dots, n$, and therefore

$$\begin{aligned} & \text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}) \\ &= \bigcup_{i=1}^n \text{GLBLABEL}(\mathcal{F}_{gen}, \{V_i\}) \\ &\leq \bigcup_{i=1}^n \text{GLBLABEL}(\mathcal{F}_{gen}, \{V'_i\}) \\ &\leq \text{GENLABEL}(\mathcal{F}_{gen}, \mathbf{V}') \quad \square \end{aligned}$$

Analogues of Theorems 3.3.3 and 3.3.6 hold for generating sets. Given a set \mathcal{F} that induces a precise labeler, a **minimal** generating set for \mathcal{F} always exists, and is guaranteed to be unique up to equivalence. Conversely, we can extend any set \mathcal{G} to an \mathcal{F} that induces a precise labeler and for which \mathcal{G} is a generating set. \mathcal{F}_{gen} is generally much smaller than either \mathcal{F} or \mathcal{F}_d , although of course it only exists under the two restrictions we outlined above.

Example 3.3.12. Continuing with Example 3.3.4, suppose \mathcal{U} is decomposable under the equivalent view ordering \leq ; this is true for instance if we take \mathcal{U} to be the set of all *single-atom* queries over `Contacts`. In this case, the set $\mathcal{F}_{gen} = \{\{V_3\}, \{V_6\}, \{V_7\}, \{V_8\}\}$ is a generating set for a \mathcal{F} that induces a precise labeler over \mathcal{U} . The size of \mathcal{F}_{gen} is now only linear in the number of attributes of `Contacts`.

The takeaway is that if \mathbf{U} is decomposable and we desire a precise labeler, it is easy to label queries using a set of security views \mathbf{S} . We can simply use the set $\{\{S_i\} \mid S_i \in \mathbf{S}\}$, consisting of singleton sets containing each view in \mathbf{S} , as our \mathcal{F}_{gen} and run algorithm `GENLABEL` to perform a labeling.

3.4 Labeling Conjunctive Queries

We now use the theory and results from Sections 3.2 and 3.3 and show how to label a particular class of queries. We focus on labeling conjunctive queries under set semantics. Our disclosure order is based on equivalent view rewriting. We restrict our attention to single-atom security views due to technical limitations that will be discussed in Section 3.4.2. Although not all of the security views that would be useful for real-world systems can be modeled without joins, a large fraction can.

We write \mathbf{U}_{atom} to denote the set of single-atom conjunctive views defined over a given database schema, and \mathbf{U}_{cv} to denote the set of all conjunctive views. Let \leq be the equivalent view rewriting order, and assume we have a set of single-atom security views \mathbf{S} . We explain how to label arbitrary conjunctive queries with subsets of \mathbf{S} .

The results of this section rely heavily on the observation that equivalent view rewriting induces a disclosure order. We verify this fact before proceeding further.

Proposition 3.4.1. *Equivalent view rewriting induces a disclosure order.*

Proof. We verify each of the disclosure order axioms in turn.

(i) If $\mathbf{V} \subseteq \mathbf{V}'$ then $\mathbf{V} \leq \mathbf{V}'$.

This is trivial, as each view in \mathbf{V} can be rewritten using itself.

(ii) If $\mathbf{V} \leq \mathbf{V}'$ and $\mathbf{V}' \leq \mathbf{V}''$ then $\mathbf{V} \leq \mathbf{V}''$.

Let $V \in \mathbf{V}$. Then there must be a rewriting R using \mathbf{V}' whose expansion R_+ is homomorphic to V . Similarly, each view $V' \in \mathbf{V}'$ that is referenced by R must have a rewriting R' using \mathbf{V}'' whose expansion R'_+ is homomorphic to V' . By replacing each body atom of R with the body of the corresponding R' and performing an appropriate substitution on distinguished variables, we obtain a rewriting of V using \mathbf{V}'' , so that $\{V\} \leq \mathbf{V}''$. Since this holds for all $V \in \mathbf{V}$, it follows that

$$\mathbf{V} = \bigcup_{V \in \mathbf{V}} \{V\} \leq \mathbf{V}''$$

as desired.

(iii) If $\mathbf{V} \leq \mathbf{V}''$ and $\mathbf{V}' \leq \mathbf{V}''$ then $\mathbf{V} \cup \mathbf{V}' \leq \mathbf{V}''$.

By hypothesis, each view in \mathbf{V} can be rewritten using \mathbf{V}'' , and each view in \mathbf{V}' can also be rewritten using \mathbf{V}'' . It follows that any view in $\mathbf{V} \cup \mathbf{V}'$ can be rewritten using \mathbf{V}'' , and therefore $\mathbf{V} \cup \mathbf{V}' \leq \mathbf{V}''$ as well. \square

We find it useful to work with a modified representation of conjunctive queries where we associate each query with a list of its body atoms and discard the head. To keep track of which of the variables are distinguished and which are existential, we tag them accordingly. For example, the query Q_2 from Figure 3.1 is represented as $[M(x_d, y_e), C(y_e, w_e, \text{'Intern'})]$, where the subscripts e and d denote existential and

distinguished variables respectively and **M** and **C** abbreviate **Meeting** and **Contact** respectively.

We present the process of labeling in two stages. First, we explain how sets of single-atom queries may be labeled. We then extend the process to sets of multi-atom queries.

3.4.1 Single-Atom Case

We will show in this section that \mathbf{U}_{atom} is decomposable. Consequently, the discussion and labeling algorithm from Section 3.3.2 apply directly. The set $\{\{S_i\} \mid S_i \in \mathbf{S}\}$, composed of singleton sets containing each of the security views, serves as a generating set for the labeler. For a complete end-to-end labeling algorithm, we only need to define implementations of the two subroutines introduced at the beginning of Section 3.3. The first determines, given $\mathbf{V}, \mathbf{V}' \subseteq \mathbf{U}_{atom}$, whether $\mathbf{V} \leq \mathbf{V}'$. The second computes the GLB function – that is, given \mathbf{V}, \mathbf{V}' it finds a \mathbf{V}'' such that $(\Downarrow \mathbf{V}) \sqcap (\Downarrow \mathbf{V}') = (\Downarrow \mathbf{V}'')$.

Determining whether $\mathbf{V} \leq \mathbf{V}'$ can be done using standard techniques from the literature on equivalent view rewriting, such as Compton’s algorithm [24]. When \mathbf{V} is a set of single-atom views, the following criterion is both necessary and sufficient.

Theorem 3.4.2 (Ordering Single-Atom Views). *$\mathbf{V} \leq \mathbf{V}'$ precisely when, for each $V \in \mathbf{V}$, there exists $V' \in \mathbf{V}'$ such that there is a homomorphism θ such that (a) $\theta V' = V$, (b) θ maps existential variables to existential variable, (c) if x is an existential variable and $y \neq x$ then $\theta y \neq \theta x$, and (d) θ maps constants to themselves.*

Proof of Theorem 3.4.2. Suppose $\mathbf{V} \leq \mathbf{V}'$. Then for every single-atom view $V \in \mathbf{V}$ there exists a conjunctive rewriting R' over the views in \mathbf{V}' whose expansion R'_+ is equivalent

to V , with homomorphisms $\theta : V \rightarrow R'_+$ and $\theta' : R'_+ \rightarrow V$. Since V is a single-atom query which is homomorphic to R'_+ , we may assume without loss of generality (by eliminating redundant body atoms) that R' is a single-atom query as well, and therefore there is a single-atom view $V' \in \mathbf{V}'$ such that $\{V\} \preceq \{V'\}$.

We now know that, by performing a substitution on the distinguished variables of V' , we can obtain a query R'_+ that is homomorphic to V . Such a substitution cannot force an equality constraint between an existential variable of R'_+ and any other variable, so (b) and (c) must hold. Every variable that is distinguished in R' must also be distinguished in R'_+ , so constraint (a) is also satisfied. And finally, a substitution on distinguished variables cannot affect the constants of R'_+ , so that constraint (d) holds.

On the other hand, query obtained by performing a substitution on the distinguished variable of R'_+ , and which therefore leaves the remaining variables unchanged, immediately satisfies conditions (a), (b), and (d). Condition (c) immediately follows from the fact that substitution is defined in a manner that prevents capture of existential variables. \square

Given single-atom views V and V' , there is only one possible choice of θ that maps the unique body atom of V' to that of V . To determine whether $\{V\} \preceq \{V'\}$, it suffices to check whether θ satisfies the conditions of Theorem 3.4.2; this can be done linear time.

More generally, if \mathbf{V} and \mathbf{V}' are sets of single-atom views then the preceding Theorem tells us that $\mathbf{V} \preceq \mathbf{V}'$ if and only for every $V \in \mathbf{V}$ there is some $V' \in \mathbf{V}'$ such that $\{V\} \preceq \{V'\}$; this check can be performed in $O(|\mathbf{V}| \cdot |\mathbf{V}'|)$ time.

We next turn our attention to the problem of computing Greatest Lower Bounds, or GLBs. Our approach is based on a procedure `GLBSINGLETON` for computing the GLB of two singleton sets of views $\{V\}$ and $\{V'\}$; this can be extended to multi-element sets of views in a manner to be explained shortly.

GLBSINGLETON is based on the idea of unification. It begins by computing a generalized *Most General Unifier* [12], or MGU, of the the bodies of V and V' . This is computed by a subroutine called GENMGU, which differs from a standard MGU computation in three ways. First, if the algorithm attempts to unify a constant with an existential variable, the unification fails. Second, if the algorithm attempts to unify an existential variable with an existential or distinguished variable, the result is an existential variable. Third, if the algorithm attempts to unify two distinguished variables, the result is another distinguished variable. We explain these differences using some examples.

First, we show why the unification of a constant with an existential variable must fail.

Example 3.4.3. Consider the following Boolean views:

$$V_{13}() :- \text{M}(9, \text{'Jim'}) \qquad V_{14}() :- \text{M}(x, y)$$

The first view tests whether `Meeting` contains a particular tuple and the second checks whether it contains any tuples at all. The standard MGU of the body atoms is equal to the first atom, but the actual GLB of the views should be \perp . There is no single-atom query that can be rewritten in terms of V_{13} and also in terms of V_{14} . This is a consequence of conditions (b) and (d) in Theorem 3.4.2.

Next, we illustrate the reasons for our handling of existential and distinguished variables.

Example 3.4.4. Consider views V_6 and V_7 from Figure 3.4:

$$V_6(x, y) :- \text{C}(x, y, z) \qquad V_7(x, z) :- \text{C}(x, y, z)$$

In our new representation they become $[\text{C}(x_d, y_d, z_e)]$ and $[\text{C}(x_d, y_e, z_d)]$ respectively. Their GENMGU is $[\text{C}(x_d, y_e, z_e)]$, i.e. V_9 from Figure 3.4. This makes intuitive sense as V_9 , the projection on the first attribute of `Contact`, accurately represents the overlap between V_6 and V_7 , i.e. the information that can be computed from either V_6 or V_7 in isolation.

Once GENMGU is available, an extra check is needed to rule out some corner cases as shown in the next example.

Example 3.4.5. Consider the following Boolean views:

$$V_{14}() :- \mathbf{M}(x, y) \qquad V_{15}() :- \mathbf{M}(z, z)$$

The GENMGU of the body atoms is $[\mathbf{M}(w_e, w_e)]$, but the GLB should be \perp by the same reasoning as in example 3.4.3. This is a consequence of condition (c) of Theorem 3.4.2.

The check to eliminate such cases is conceptually straightforward. It involves finding situations where computing GENMGU forces a new equality constraint on two values in the same original atom, and where at least one of these values was an existential variable. If we find such a situation or if GENMGU fails, GLBSINGLETON returns \perp ; otherwise it returns the output of GENMGU.

We next verify the correctness of the process outlined above.

Theorem 3.4.6. $V^S = \text{GLBSINGLETON}(V, V')$ satisfies $\Downarrow \{V\} \sqcap \Downarrow \{V'\} = \Downarrow \{V''\}$.

Proof Sketch. GLBSINGLETON attempts to compute the Most General Unifier of V and V' . However, it prevents each existential variable in V from being unified with any other variable or constant. This ensures that V^S can be obtained by performing a substitution on the distinguished variables in V , and therefore $\{V^S\} \leq \{V\}$, so that $\Downarrow \{V^S\} \leq \Downarrow \{V\}$. An analogous argument shows that $\Downarrow \{V^S\} \leq \Downarrow \{V'\}$. It follows that

$$\Downarrow \{V^S\} \leq \Downarrow \{V\} \sqcap \Downarrow \{V'\}$$

For the other direction, suppose that $\{V''\} \leq \{V\}$ and $\{V''\} \leq \{V'\}$ for some $V'' \in \mathbf{U}_{atom}$. Then there must be a substitution on the distinguished variables in V (resp. V') that yields

an atom that is isomorphic to V'' . In particular, if there is a constant or existential variable at some position in the body of V (resp. V') then there must be a constant or existential variable at the same position in V'' . Similarly, if the terms at two different positions are equal in V (resp. V') then the terms at the same positions in V'' must also be equal.

This means that V^S and V'' are isomorphic on the existential variables and constants of V^S . The remaining variables in V^S are all distinguished, so by performing a substitution on the distinguished variables of V^S , we can obtain a view that is isomorphic to V'' , and therefore $\{V''\} \leq \{V^S\}$. It follows that

$$\Downarrow \{V\} \sqcap \Downarrow \{V'\} \leq \Downarrow \{V^S\}$$

and therefore the two must be equal, completing the proof. \square

GLBSINGLETON can be extended to non-singleton sets for a complete implementation of $GLB(\mathbf{V}, \mathbf{V}')$. We simply compute the pairwise GLBSINGLETON of singleton sets containing each pair of views $V \in \mathbf{V}, V' \in \mathbf{V}'$ and union all the results together. This completes the description of GLB, giving us the last tool we need to label queries using the techniques from Section 3.3.2.

Given $\mathbf{V}, \mathbf{V}' \subseteq \mathbf{U}_{atom}$, a set $\mathbf{V}'' \subseteq \mathbf{U}_{atom}$ such that $(\Downarrow \mathbf{V}) \sqcap (\Downarrow \mathbf{V}') = (\Downarrow \mathbf{V}'')$ can be computed as follows:

- 1: **procedure** GLB(\mathbf{V}, \mathbf{V}')
- 2: $\mathbf{V}'' \leftarrow \emptyset$
- 3: **for** each view $V \in \mathbf{V}$ **do**
- 4: **for** each view $V' \in \mathbf{V}'$ **do**
- 5: **if** GLBSINGLETON(V, V') $\neq \perp$ **then**
- 6: $\mathbf{V}'' \leftarrow \mathbf{V}'' \cup \{\text{GLBSINGLETON}(V, V')\}$
- 7: **end if**

8: **end for**
9: **end for**
10: **return V''**
11: **end procedure**

In order to verify the correctness of this procedure, we must first show that \mathbf{U}_{atom} is *decomposable*, in the sense discussed above.

Proposition 3.4.7. *Suppose that $\{V\} \leq \mathbf{V} \cup \mathbf{V}'$, where $V \in \mathbf{U}_{atom}$ and $\mathbf{V}, \mathbf{V}' \subseteq \mathbf{U}_{atom}$. If $\{V\} \leq \mathbf{V} \cup \mathbf{V}'$ then either $\{V\} \leq \mathbf{V}$ or else $\{V\} \leq \mathbf{V}'$.*

Proof. Suppose $\{V\} \leq \mathbf{V} \cup \mathbf{V}'$. Then there must be a rewriting R using the views in $\mathbf{V} \cup \mathbf{V}'$ that is homomorphic to V . Since V contains exactly one body atom, we may assume WLOG (by folding R_+ if needed) that the expansion R_+ of R contains exactly one body atom. If this body atom originates from \mathbf{V} then $\{V\} \leq \mathbf{V}$. On the other hand, if the body atom originates from \mathbf{V}' then $\{V\} \leq \mathbf{V}'$. \square

We next verify the correctness of the GLB procedure shown above.

Theorem 3.4.8. *GLB(\mathbf{V}, \mathbf{V}') returns a set \mathbf{V}^S such that $(\Downarrow \mathbf{V}) \sqcap (\Downarrow \mathbf{V}') = (\Downarrow \mathbf{V}^S)$.*

Proof. \mathbf{V}^S is a set of views of the form $\text{GLBSINGLETON}(V, V')$ where $V \in \mathbf{V}$ and $V' \in \mathbf{V}'$. For each pair of views we have

$$\{\text{GLBSINGLETON}(V, V')\} \leq \{V\} \leq \mathbf{V} \leq \Downarrow \mathbf{V}$$

and $\{\text{GLBSINGLETON}(V, V')\} \leq \{V'\} \leq \mathbf{V}' \leq \Downarrow \mathbf{V}'$

so that

$$\{\text{GLBSINGLETON}(V, V')\} \leq (\Downarrow \mathbf{V}) \sqcap (\Downarrow \mathbf{V}')$$

Hence

$$\Downarrow \mathbf{V}^S \equiv \bigcup_{V \in \mathbf{V}, V' \in \mathbf{V}'} \Downarrow \{\text{GLBS}_{\text{SINGLETON}}(V, V')\} \leq (\Downarrow \mathbf{V}) \sqcap (\Downarrow \mathbf{V}')$$

In the other direction, suppose $\mathbf{V}'' \leq (\Downarrow \mathbf{V}) \sqcap (\Downarrow \mathbf{V}')$, and let $V'' \in \mathbf{V}''$. Then $\{V''\} \leq (\Downarrow \mathbf{V})$, so there must be some $V \in \mathbf{V}$ such that $\{V''\} \leq \{V\}$ because \mathbf{U}_{atom} is decomposable. Similarly, $\{V''\} \leq (\Downarrow \mathbf{V}')$, so there must be some $V' \in \mathbf{V}'$ such that $\{V''\} \leq \{V'\}$. It follows that $\{V''\} \leq \{\text{GLBS}_{\text{SINGLETON}}(V, V')\} \leq (\Downarrow \mathbf{V}^S)$. Taking the union over all $V'' \in \mathbf{V}''$, we conclude that

$$\mathbf{V}'' = \bigcup_{V'' \in \mathbf{V}''} \{V''\} \leq (\Downarrow \mathbf{V}^S)$$

and therefore \mathbf{V}^S is a greatest lower bound, as required. \square

3.4.2 Multi-Atom Case

The set \mathbf{U}_{cv} of arbitrary conjunctive queries is not in general decomposable; therefore, we are unable to use the same techniques as above. However, because we have restricted the set \mathbf{S} to contain single-atom views only, we can perform labeling efficiently by solving the problem in two steps. To label a set of queries $\mathbf{Q} \subseteq \mathbf{U}_{\text{cv}}$, we first convert each $Q \in \mathbf{Q}$ into a set of single-atom queries using the `DISSECT` algorithm described below. In the second step, we compute the disclosure label of the resulting set of single-atom views using the algorithm discussed in the previous subsection.

The `DISSECT` algorithm begins by computing a *folding* [19] of Q , which intuitively removes “redundant” atoms from Q . Next, it splits up the folding of Q into its constituent atoms, except that any existential variable that appears in at least two atoms is promoted to a distinguished variable.

Example 3.4.9. Consider query Q_2 from Figure 3.1, i.e. $[M(x_d, y_e), C(y_e, w_e, \text{'Intern'})]$. The result of running DISSECT on this query is a set that contains two single-atom queries: $[M(x_d, y_d), C(y_d, w_e, \text{'Intern'})]$.

Intuitively, the reason we need to promote existential variables to distinguished ones is that we are labeling with single-atom views. Recall that the set of single-atom views in a query's disclosure label must contain enough information to uniquely determine the query's answer. Any set of single atom security views that allows a join to be computed must reveal the values of the join attributes.

We can show that DISSECT is a disclosure labeler with domain $\wp(\mathbf{U}_{cv})$ and image $\wp(\mathbf{U}_{atom})$. As the composition of two labelers is also a labeler, we can create a disclosure labeler for multi-atom conjunctive queries by combining DISSECT with our single-atom labeling procedure.

Proposition 3.4.10. *DISSECT is a disclosure labeler.*

Proof. We verify each of the disclosure labeler axioms in turn:

- (i) If $\mathbf{V} \subseteq \mathbf{U}_{cv}$ then $\text{DISSECT}(\mathbf{V}) \equiv \mathbf{V}'$ for some $\mathbf{V}' \in \mathbf{U}_{atom}$.

The last step of DISSECT ensures that the procedure always returns a set of single-atom queries as its output.

- (ii) If $\mathbf{V} \subseteq \mathbf{U}_{atom}$ then $\text{DISSECT}(\mathbf{V}) \equiv \mathbf{V}$.

For each view $V \in \mathbf{V}$, the DISSECT procedure folds V , promotes any existential variable that appears in two distinct body atoms to a fresh distinguished variable, and then generates a single-atom view for each body atom of V . If $V \in \mathbf{U}_{atom}$ then each of these steps leaves V unchanged. This means that each element if \mathbf{V} is equivalent to an element of $\text{DISSECT}(\mathbf{V})$ and vice versa.

(iii) If $\mathbf{V} \subseteq \mathbf{U}_{cv}$ then $\mathbf{V} \leq \text{DISSECT}(\mathbf{V})$.

For each view $V \in \mathbf{V}$, folding V yields a view V' such that $\{V\} \equiv \{V'\}$, and therefore $\{V\} \leq \{V'\}$. Promoting existential variables to distinguished variables yields a new view V'' , and by performing a substitution on the distinguished variables of V'' , we can obtain a view that is isomorphic to V' . This means that $\{V'\} \leq \{V''\}$. The final step yields a set of single-atom views \mathbf{V}''' ; by joining together all the views in \mathbf{V}''' , we can obtain a view that is isomorphic to V'' , and therefore

$$\{V\} \leq \{V'\} \leq \{V''\} \leq \mathbf{V}''' \leq \text{DISSECT}(\mathbf{V})$$

It follows that

$$\mathbf{V} = \bigcup_{V \in \mathbf{V}} \{V\} \leq \text{DISSECT}(\mathbf{V})$$

as desired.

(iv) If $\mathbf{V}, \mathbf{V}' \subseteq \mathbf{U}_{cv}$ and $\mathbf{V} \leq \mathbf{V}'$ then $\text{DISSECT}(\mathbf{V}) \leq \text{DISSECT}(\mathbf{V}')$.

Let $V \in \mathbf{V}$, so that $\{V\} \leq \mathbf{V}'$. By folding V we obtain a view V' such that $\{V\} \equiv \{V'\}$. Then $\{V'\} \leq \{V\} \leq \mathbf{V}'$, so there must be a rewriting R using \mathbf{V}' whose expansion R_+ is homomorphic to V' . Since V' is folded, any existential variable that appears in two distinct atoms of V' must also appear in two distinct atoms of R_+ , and therefore the corresponding variables in \mathbf{V}' are either distinguished or else appear in multiple atoms of the corresponding view in \mathbf{V}' ; in the latter case, they will be promoted to fresh distinguished variables. In either case, the updated V' can be answered using the updated \mathbf{V}' . This means that each atom in the body of the updated V' is isomorphic to an atom of a view in the updated \mathbf{V}' (after performing an appropriate substitution on distinguished variables), and therefore $\text{DISSECT}(\{V\}) \leq \text{DISSECT}(\mathbf{V}')$.

It follows that

$$\text{DISSECT}(\mathbf{V}) = \bigcup_{V \in \mathbf{V}} \text{DISSECT}(\{V\}) \leq \text{DISSECT}(\mathbf{V}')$$

as desired. □

3.5 Implementation

At this point, we have introduced our new notion of a disclosure labeler that is data-derived, semantically meaningful and expressive, and we have presented practical algorithms for labeling conjunctive queries. In this section, we describe two key optimizations that allow us to efficiently manage complex security policies for regulating the cumulative disclosure of information over time. First, we store disclosure labels in a heavily compressed format that makes comparisons between different disclosure labels very fast. And second, we represent security policies in a way that allows us to make policy decisions without ever needing to refer back to a list of previously executed queries.

3.5.1 Representing Disclosure Labels

We begin by revisiting the `GLBLABEL` disclosure labeling algorithm from Section 3.3.1. In its simplest form `GLBLABEL` takes as input a set of security views \mathcal{F}_{gen} and a singleton set $\{V\}$ whose disclosure label we wish to find, and returns the GLB of the following collection of singleton sets:

$$\{\{V_i\} : V_i \in \mathcal{F}_{gen} \text{ and } \{V\} \leq \{V_i\}\}$$

In practice, however, computing the GLB is completely unnecessary. Instead, we compute

$$\ell^+(\{V\}) = \{V_i \in \mathcal{F}_{gen} : \{V\} \leq \{V_i\}\}$$

Roughly speaking, this is the set of all security views that uniquely determine the answer to V . If we know $\ell^+(\{V\})$, we can compute $\ell(\{V\})$. Furthermore, we can now efficiently compare the disclosure labels of two different points $\ell(\{V\})$ and $\ell(\{V'\})$ in the lattice of disclosure labels:

$$\ell(\{V\}) \leq \ell(\{V'\}) \text{ if and only if } \ell^+(\{V\}) \supseteq \ell^+(\{V'\})$$

We provide an example in order to solidify this idea:

Example 3.5.1. Continuing Example 3.3.12, let

$$\mathcal{F}_{gen} = \{\{V_3\}, \{V_6\}, \{V_7\}, \{V_8\}\}$$

The disclosure label for $\{V_9\}$ is $GLB(\{V_3\}, \{V_6\}, \{V_7\})$, and consequently, $\ell^+(\{V_9\}) = \{V_3, V_6, V_7\}$. Similarly, the disclosure label for $\{V_{12}\}$ is $GLB(\{V_3\}, \{V_6\}, \{V_7\}, \{V_8\})$, so that $\ell^+(\{V_{12}\}) = \{V_3, V_6, V_7, V_8\}$. Examining these two sets, it is clear that $\ell^+(\{V_{12}\}) \supseteq \ell^+(\{V_9\})$, and we therefore conclude that $\ell(V_{12}) \leq \ell(V_9)$.

In practice, these subsets of \mathcal{F}_{gen} can be represented as bit vectors; we use bit mask operations to determine whether one subset contains another. Since $\{V_1\} \leq \{V_2\}$ only if V_1 and V_2 are views over the same base relation, we can further optimize this representation. In our current implementation, the low 32 bits of a 64-bit integer track which base relation a view corresponds to, and the remaining 32 bits represent the elements of \mathcal{F}_{gen} that are associated with that relation. In this way, a single 64-bit integer can store a disclosure label for a disclosure lattice with up to 2^{32} distinct relations, each of which is associated with 32 distinct elements from \mathcal{F}_{gen} . There is nothing special about the number 32, and the representation can easily be generalized to any number of bits.

We extend this representation to multi-atom disclosure labels by using arrays of single-atom disclosure labels. For instance, in Example 3.5.1, the disclosure label of $\{V_6, V_7\}$ can be stored as a two-element array whose first element is $\ell^+(\{V_6\})$ and whose second element is $\ell^+(\{V_7\})$.

Complexity Analysis: Let n denote the number of atoms in the input query, and m denote the number of security views. The DISSECT algorithm from Section 3.4.2 relies on query folding as a subroutine. Query folding is known to be NP-hard, and our

current implementation uses a brute-force search that runs in time that is exponential in n . Dissection yields a set of at most n single-atom views, and in the worst case, the labeler must determine whether each of the n views returned by `DISSECT` can be rewritten in terms of each of the m security views, for a total of $O(n \cdot m)$ comparisons. Each comparison can be performed in expected linear time relative to the total size of its two input atoms. Once they have been computed, the disclosure labels of an r -atom query and an s -atom query can be compared in time $O(r \cdot s)$.

3.5.2 Representing Security Policies

Formally, a security policy is defined in Section 3.2.4 as a subset of the elements in a labeler's disclosure lattice. However, as we have already noted, disclosure lattices can become enormous even for small databases; storing security policies explicitly is therefore impractical. In this section, we discuss a different representation of security policies that drastically reduces space consumption. As an added bonus, we are able to track and restrict cumulative disclosure with very little space or computational overhead. We restrict our discussion to a system with a single principal; a generalization to multiple principals is straightforward.

Let us first consider the simpler problem of enforcing a stateless security policy. When a principal issues a query Q , a reference monitor decides whether to answer or refuse the query based solely on the query's disclosure label and on the security policy itself. In this model, a security policy can be represented as a set \mathbf{V} of disclosure labels for single-atom views; a query is answered if its disclosure label is below \mathbf{V} , and is refused otherwise.

We now discuss a variant of this algorithm that limits cumulative information dis-

closure over time. When a principal issues a query Q_n , a reference monitor looks at both Q_n and the list of previously answered queries Q_1, Q_2, \dots, Q_{n-1} . In the case where $\{Q_1, Q_2, \dots, Q_n\} \leq \mathbf{V}$, the query is answered. Otherwise, the query is refused.

Crucially but perhaps counterintuitively, the two models described above are actually equivalent for stateless policies. Formally, the first model ensures that $\{Q_i\} \leq \mathbf{V}$ for each $i = 1, 2, \dots, n$. The second model ensures that $\{Q_1, Q_2, \dots, Q_n\} \leq \mathbf{V}$. Equivalence follows immediately from the definition of a disclosure order (Definition 5.1.6).

What this means in practice is that even a stateless reference monitor can restrict cumulative information disclosure. Unfortunately, this guarantee comes at a cost: it is no longer possible to represent *stateful* security policies, such as the *Chinese Wall* policies required by many business applications, with this model. In order to support such policies, we represent a security policy as a *collection* of sets of single-atom disclosure labels, say $\{\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_k\}$. We enforce the invariant that if Q_1, Q_2, \dots, Q_n are the queries that have been answered so far then $\{Q_1, Q_2, \dots, Q_n\} \leq \mathbf{V}_i$ must hold for *some* \mathbf{V}_i . We refer to each \mathbf{V}_i as a *partition* of the security policy.

Example 3.5.2. Consider the security policy $\{\mathbf{V}_1, \mathbf{V}_2\}$ in which $\mathbf{V}_1 = \{V_1\}$ and $\mathbf{V}_2 = \{V_3\}$. This policy encodes the constraint that a principal may access either `Meeting` or `Contact`, but not both. If the principal Alice issues query V_6 , this query will be accepted, since $\{V_6\} \leq \mathbf{V}_2$. If she next issues the query V_7 , this query will also be accepted, since $\{V_6, V_7\} \leq \mathbf{V}_2$. However, if she then issues the query V_2 , this query will be refused, since $\{V_6, V_7, V_2\} \not\leq \mathbf{V}_1$ and $\{V_6, V_7, V_2\} \not\leq \mathbf{V}_2$.

A naïve implementation of this scheme would require us to search through a principal's entire query history whenever we make a policy decision for a new query. Fortunately, this is not necessary. In fact, we only need to keep track of which of the \mathbf{V}_i are consistent with all the queries answered so far; we can do so with a bit vector that

Attribute	FQL Permissions	Graph API Permissions	Correct Labeling
pic (“picture” in Graph API)	none	any for pages with whitelisting/targeting restrictions, otherwise none	FQL
timezone	any	Available only for the current user	Graph API
devices	any	any; only available for friends of the current user	Graph API
relationship_status	any	user_relationships or friends_relationships	Graph API
quotes	user_likes or friends_likes	user_about_me or friends_about_me	FQL
profile_url (“link” in Graph API)	any	none	FQL

In the table above, “any” means any nonempty set of permissions and “none” means no permissions are required

Table 3.2: Inconsistencies between the FQL and Graph API permissions labeling of User attributes.

contains one bit for each partition of the policy.

Example 3.5.3. In Example 3.5.2, the reference monitor’s bit vector is initially $\langle 1, 1 \rangle$, which indicates that $\emptyset \leq \mathbf{V}_1$ and $\emptyset \leq \mathbf{V}_2$. After answering V_6 , the bit vector becomes $\langle 1, 0 \rangle$ because the $\{V_6\} \leq \mathbf{V}_1$ but $\{V_6\} \not\leq \mathbf{V}_2$. The bit vector is left unchanged after the second query. If the third query was answered, the bit vector would become $\langle 0, 0 \rangle$ to indicate that $\{V_6, V_7, V_2\}$ is not below either \mathbf{V}_1 or \mathbf{V}_2 in the lattice of disclosure labels. However, the reference monitor will instead refuse the query and leave the bit vector as $\langle 1, 0 \rangle$.

3.6 Labeling in Practice

In this section, we showcase the practical applicability and usefulness of our labeling-based disclosure control techniques. We begin by presenting results of a manual review

of an existing labeling-based disclosure control system – the permissions structure associated with Facebook’s Graph API and FQL. Next, we present experimental results from an implementation of our labeling algorithms. We conclude by evaluating the throughput of the policy checker described in Section 3.5.2.

3.6.1 Reviewing Facebook’s APIs

Facebook provides two APIs through which apps can query user data: the Graph API and FQL. They also define a set of permissions such as `user_likes` and `friends_likes`, each of which grants an app access to a particular view over the data. Before an app can issue an API query, it must request access to a specific set of permissions. In our terminology, a set of permissions corresponds to a *disclosure label*, and each Facebook app is effectively a different principal.

Facebook’s developer documentation specifies the minimal set of permissions needed to execute different API queries. In other words, it provides a hand-generated disclosure label for each of these queries. We hypothesized that as the APIs grow larger and more complex, manual labeling becomes error-prone. We identified 42 different views over the User table accessible through both APIs and compared the respective permissions as given in the documentation. That is, we identified pairs of corresponding queries in both APIs where both queries selected a particular attribute of the User table. We then compared the required permissions listed in the documentation for each pair of queries.

We found discrepancies in the permissions needed for six of the 42 views; details are shown in Figure 3.2. This illustrates the difficulty of manually labeling queries. Our findings are consistent with previous studies of human-generated query labels in a different setting, namely Android apps [30].

For each of the six Facebook views mentioned above, we issued appropriate queries in both APIs to determine which permissions were really required. In all six cases, we found that the same query in both APIs required the same permissions, as shown for each query in the last column of Figure 3.2. Thus, the inconsistencies were in the documentation only. Nonetheless, such errors are alarming. Tracking complex permission structures by hand is challenging, and the chances that developers will select the wrong permissions for their apps are compounded if they rely on inaccurate documentation.

3.6.2 Experimental evaluation

We implemented and evaluated two key systems: a disclosure labeler for multi-atom queries, and a mechanism that makes policy decisions based on the labeler’s output. The disclosure labeler, which emphasized scalability but not raw performance, was implemented in Java, and was tested with the Java 1.7 VM. The policy mechanism was implemented in C and compiled with GCC 4.2. All our tests were conducted on a laptop with a 2.9GHz Intel Core i7 processor running Mac OS X 10.8. Our benchmarks measured process rather than wall time.

Our test database contained eight different relations that captured core functionality from the Facebook API. The largest of these was the User relation, which contained 34 distinct attributes. Each of the remaining relations contained between 3 and 10 attributes.⁴

For each relation, we selected a set of security views that could support the confidentiality policies described in Facebook’s developer documentation. The most complex relation, the User relation, required us to define a generating set \mathcal{F}_{gen} with 16 distinct

⁴In preliminary tests on synthetic data, we tried increasing the total number of relations to 1,000 while keeping the number of security views per relation constant; the total number of relations did not have any appreciable impact on the hash-based disclosure labelers’ throughput.

security views; most of the other relations we considered could be modeled using just three views. The main difficulty that we encountered was that some of the permissions that Facebook uses require a notion of joins. For instance, there is a permission that allows a Facebook app to see the birthdays of all of a user’s Facebook friends. Formally, this can be modeled using a join between the `User` relation and the `Friend` relation. The implementation we used did not support security views with joins in them. We dealt with this issue by adding an extra column to each relation that indicated whether the owner of a given tuple was friends with the principal executing the query. Since the list of a user’s friends is available to any app running on behalf of that user, this denormalization did not affect the accuracy of our model.

After examining a number of sample Facebook applications, we decided to use a workload of queries that were randomly generated with the following process:

- (i) Select a random relation from the schema.
- (ii) Select a random subset of its attributes.
- (iii) Randomly request these attributes for either (i) the current user, (ii) friends of the current user, (iii) friends of friends of the current user, or (iv) a non-friend.

In Step (3) above, we note that Option (ii) involved a join with the `Friend` relation, and Option (iii) involved *two* joins with the `Friend` relation. Hence, each query contained between one and three body atoms. In order to stress-test our algorithm, we extended our workload to generate (unrealistically) complex queries; we did this by repeating the process above between one and five times, and joining the resulting subqueries on the `uid` (User ID) attribute, which appeared in all the relations we considered.

We used this workload to evaluate the performance of three different versions of our disclosure labeling algorithm. The first version, which we used as a baseline, was a straightforward adaptation of the `LABELGEN` algorithm from Section 3.3.2. The second

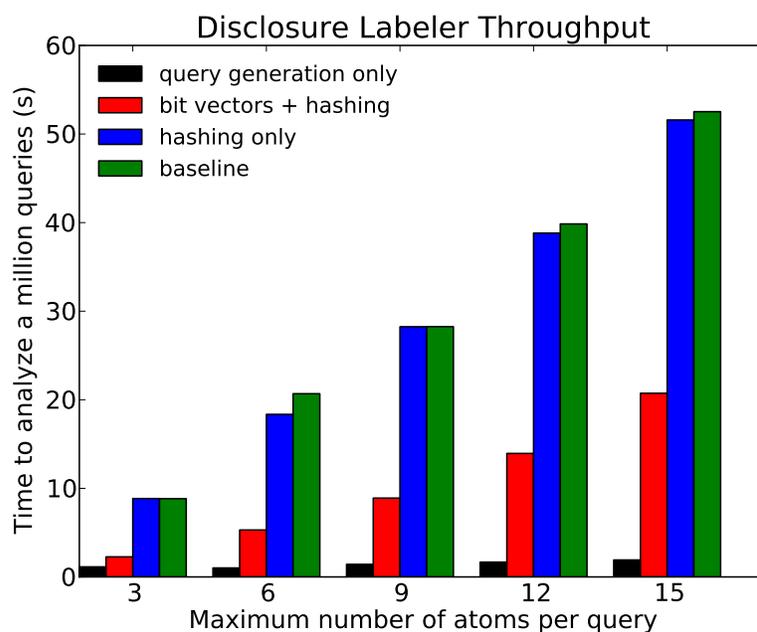
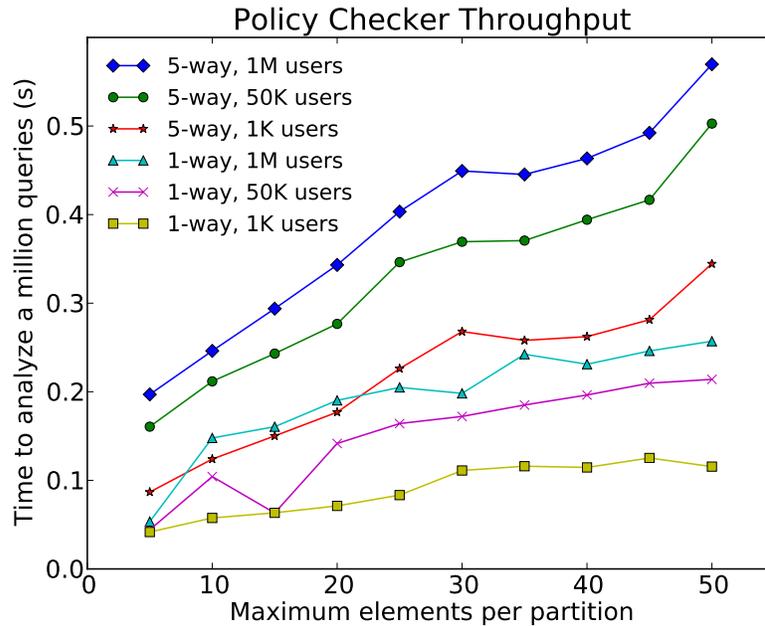


Figure 3.5: Disclosure labeler performance.

version used a hashtable to partition views based on the relation they referenced. The third version made use of both hashtable partitioning and the bit vector optimization from Section 3.5.1. In a fourth experiment, we considered only the time needed to randomly generate parsed queries but not to label them.

Results are shown in Figure 3.5. The labeler that only incorporated hashing generally outperformed the baseline by a small margin. The labeler that additionally made use of the bit vector optimization consistently outperformed both of them by a factor of 3x to 4x – it was able to process a million queries, each with between 1 and 3 body atoms, in slightly more than 2 seconds.

We hypothesized that the optimizations from Section 3.5 would make it possible to reason about complex security policies very efficiently once the disclosure label for a query was known. To verify this hypothesis, we wrote a simple policy checker that maintained information about the security policies of between 1,000 and 1,000,000



The top-to-bottom ordering in the legend mirrors the top-to-bottom ordering of the curves.

Figure 3.6: Policy checker performance.

distinct principals. Each principal’s security policy was randomly generated. The maximum number of partitions per policy was set to either 1 (a stateless security policy) or 5 (a fairly complex Chinese Wall policy). However, the actual number of partitions per policy could vary between principals, reflecting the intuition that some principals would require more complex policies than others. Similarly, we allowed the maximum number of elements (i.e., single-atom views) per partition to vary between 5 and 50. Intuitively, we should expect the number of security views to increase as users define fine-grained policies to control how their data is shared.

We ran our experiment on a collection of 10 million disclosure labels output by the previous experiment. Each labeled query contained between one and three body atoms. Queries were randomly assigned to principals, and the appropriate policy was enforced for each principal.

The results are shown in Figure 3.6. For relatively simple policies, the analysis time for a million queries was between 0.04 and 0.15 seconds, depending on the number of principals in the system. Throughput decreased very gradually as the number of principals in the system increased. This decrease was likely due to issues with cache locality: as the number of principals grew larger, it became increasingly improbable that the metadata for a randomly selected principal would reside in an on-chip cache. For a million principals, our system was able to analyze a million disclosure labels in about 0.57 seconds on the most complex policies we tested.

CHAPTER 4

DISCLOSURE LABELING ON MULTIPLICITY-SENSITIVE QUERIES

In Chapter 3 we proposed a theory of *disclosure labeling* that provided a formal foundation for reasoning about the information needed to answer a database query. The basic idea was to measure the information needed to answer a given query in terms of a set of *security views* that were defined by a human administrator and revealed known types of information about the dataset.

In this Chapter we generalize the results of Chapter 3 to a language that supports bag, bag-set, and set semantics. The difference between the three can be summarized as follows: bag semantics preserve duplicate elements in *both* a query’s input and its output, while set semantics ignore duplicates in a query’s input and eliminate duplicates from a query’s output. Bag-set semantics preserve duplicate elements in a query’s output but eliminate duplicates from input relations as an initial preprocessing step.

For example, consider the query

$$Q(n) :- \text{User}(u, n)$$

Under the set semantics used in Chapter 4, any name can appear at most once in the query’s answer, no matter how many times it appears in the input database. These semantics can be achieved in SQL using the `DISTINCT` keyword as follows:

```
SELECT DISTINCT name FROM User
```

The input database might contain ten different users named “John Doe,” but the tuple (‘ John Doe ’) will appear only once in the answer to Q when evaluated under set semantics. However, it will appear ten times if Q is instead evaluated under bag semantics. Evaluation of Q under bag semantics corresponds to the SQL query

```
SELECT name FROM User
```

User		Friend	
UID	Name	UID1	UID2
1	Babbage, Charles	1	3
2	Church, Alonzo	3	1
3	Lovelace, Ada	2	4
4	Turing, Alan	4	2

Figure 4.1: Sample dataset based on the Facebook Apps schema.

Finally, queries evaluated under bag-set semantics retain duplicate tuples in the query answer but not in the input database. If Q is evaluated under bag-set semantics on a database that contains two copies of the tuple (1, 'John Doe') then the tuple ('John Doe') will *once* in the query's answer. However, if Q is evaluated on a database that contains the tuples (1, 'John Doe') and (2, 'John Doe') then ('John Doe') will appear *twice* in the query's answer. Evaluation of Q under bag-set semantics corresponds to the following SQL query:

```
SELECT name FROM User GROUP BY uid, name
```

Finally, there are SQL queries such as

```
SELECT name FROM User WHERE uid IN
(SELECT uid2 FROM Friend WHERE uid1 = 4)
```

that use a mix of bag, bag-set, and set semantics. The outer query in the example above is evaluated under bag semantics, while the inner subquery is evaluated under set semantics.

The remainder of the Chapter is laid out as follows. In Section 4.1 we define a language that generalizes conjunctive queries under set, bag-set, and bag semantics. In Section 4.2 we show how to check for equivalent rewritings of queries using views in this language. Finally, in Section 4.3 we show how to compute disclosure labels for queries in this new language.

4.1 Core Query Language: Definition and Formal Semantics

We begin by defining a query language that generalizes conjunctive queries under set, bag-set, and bag semantics. The language is similar to one proposed by Cohen [23], but with three key changes. First, we tweak Cohen’s definition in order to ensure that conjunctive queries map datasets to datasets; this allows us to use the output of one query as the input to another. Second, we restrict Cohen’s language to ensure that the composition of two conjunctive queries can itself be expressed as a conjunctive query in the same language. Third, we model the distinction between bag-set and bag semantics using a slightly different formalism than Cohen. This allows us to capture certain nuances of query equivalence that Cohen’s formalism does not. For instance, the query

```
SELECT U1.name FROM User U1
```

returns the same answer as the query

```
SELECT U1.name FROM User U1
WHERE U1.uid IN (SELECT U2.uid FROM User U2)
```

on every dataset. Intuitively, the reason is that every tuple that appears in U1 also appears in U2, so the `WHERE` clause of the second SQL query is always satisfied. Our formalism makes it possible to prove this equivalence; Cohen’s does not.¹ Taken together, these changes provide us with a language that captures many crucial nuances of real SQL queries but also retains the properties that make efficient disclosure labeling possible under set semantics.

¹Formally, the issue is that Cohen’s formalism would model U1 as a copy atom and U2 as a relational atom. Query homomorphisms in Cohen’s model, which are used to characterize equivalence, cannot map relational atoms to copy atoms.

4.1.1 Datasets and Queries

We use x, y, z and i, j, k to denote variables. The former range over constants in the dataset, whereas the latter range over copy numbers. For this reason, we call the former *regular variables* (or simply *variables* for short) and we call the latter *copy variables*. We use c, d to denote constants. A term, denoted as s, t , is either a variable or a constant. A *relational atom* has the form $A(s_1, \dots, s_n)$, where A is a predicate of arity n . A dataset \mathcal{D} is a multiset containing atoms of the form $A(c_1, \dots, c_n)$ where A is a predicate of arity n and each c_i is a constant.

We add an extra *copy variable* i to each atom in a query's body; this variable allows us to keep track of duplicate tuples in the input dataset. The resulting atoms are referred to as *copy atoms*. Formally, a copy atom has the form $A(s_1, \dots, s_n; i)$ where A is a predicate of arity n . We will use the notations $A(\bar{s})$ and $A(\bar{s}; i)$ to denote relational and copy atoms respectively, where \bar{s} stands for a sequence of terms s_1, \dots, s_n .

Definition 4.1.1. A query is a non-recursive expression of the form

$$Q(\bar{t}) :- A_1(\bar{t}_1; i_1) \wedge \dots \wedge A_n(\bar{t}_n; i_n), M$$

where each $A_k(\bar{t}_k; i_k)$ is a copy atom, M is a set of variables, and the following hold:

- Every variable in \bar{t} appears as an ordinary variable in some \bar{t}_k .
- Every element of M appears in some \bar{t}_k and does not appear in \bar{t} .
- All the copy variables are distinct, and no copy variable appears in \bar{t} or in any \bar{t}_k .

We refer to variables that appear in the head of a query as *distinguished variables*, and call variables that only appear in the query body *non-distinguished variables*. Non-distinguished variables can be divided into *multiset variables* that appear in M and

set variables that do not. A query which contains only multiset and distinguished variables will be evaluated under bag semantics, while a query that contains only set and distinguished variables will be evaluated under set semantics. A query whose non-copy variables are multiset variables and whose copy variables are set variables will be evaluated under bag-set semantics. Many real queries combine elements of both bag and set semantics. In our running example, the query

```
SELECT name FROM User
```

would be expressed as

$$Q_1(n) :- \text{User}(u, n; i), \{u, i\}$$

whereas the query

```
SELECT DISTINCT name FROM User
```

would be expressed as

$$Q_2(n) :- \text{User}(u, n; i), \emptyset$$

The variables u and i are multiset variables in the former (and therefore retain duplicates) and are set variables in the latter (and therefore ignore duplicates). The SQL query

```
SELECT U1.name FROM User U1
WHERE U1.uid IN (SELECT U2.uid FROM User U2)
```

would be translated as

$$Q_3(n_1) :- \text{User}(u, n_1; i_1) \wedge \text{User}(u, n_2; i_2), \{u, n_1, i_1\}$$

Notice that all the existential variables that appear in the query's first body atom are multiset variables; this ensures that duplicate tuples from the first atom will be retained. On the other hand, the remaining existential variables are all set variables; this ensures that duplicate entries in the second atom will be ignored.

Evaluation semantics are based on *satisfying assignments*, which we define next.

UID	Name
1	John Doe
1	John Doe
2	Jane Roe

(a) Sample User table with duplicate tuples.

UID	Name	Copy
1	John Doe	1
1	John Doe	2
2	Jane Roe	1

(b) Table that distinguishes between duplicate tuples using copy variables.

Figure 4.2: Effect of copy variables on query evaluation.

Definition 4.1.2. Let γ be a mapping of the terms in a query Q to constants. We say that γ is a satisfying assignment of Q with respect to the dataset \mathcal{D} if the following hold:

- (i) γ is the identity mapping on constants.
- (ii) For each body atom $A(\bar{t}; i)$ in Q , $\gamma i \in \mathbb{N}_+$ and \mathcal{D} contains at least γi copies of $A(\gamma\bar{t})$.

The copy variables allow us to distinguish between duplicate tuples in a table. Intuitively, condition (ii) means that if \mathcal{D} contains $N > 0$ copies of the atom $A(\gamma\bar{t})$ then γi can be mapped to any of the integers $1, 2, \dots, N$. Query evaluation behaves as if a table containing duplicate tuples, as in Figure 4.2 (a) is transformed into a table without duplicates, as in Figure 4.2 (b) as a preprocessing step.

Let $\Gamma(Q, \mathcal{D})$ denote the set of satisfying assignments of Q with respect to \mathcal{D} , and let γ be an assignment of the multiset and distinguished variables in Q , denoted $M(Q)$, to constants. We say that γ is *satisfiably extendable* if there is an assignment $\gamma' \in \Gamma(Q, \mathcal{D})$ such that γ and γ' coincide on all terms for which γ is defined. We write $\Gamma_M(Q, \mathcal{D})$ to denote the set of satisfiably extendable assignments of $M(Q)$ with respect to \mathcal{D} .

For the query Q_1 defined above and the User table in Figure 4.2 (a), there are three

satisfying assignments in $\Gamma(Q_1, \mathcal{D})$:

$$\{u \mapsto 1, n \mapsto \text{'John Doe'}, i \mapsto 1\}$$

$$\{u \mapsto 1, n \mapsto \text{'John Doe'}, i \mapsto 2\}$$

$$\{u \mapsto 2, n \mapsto \text{'Jane Roe'}, i \mapsto 1\}$$

The satisfying assignments in $\Gamma(Q_2, \mathcal{D})$ are the same as those in $\Gamma(Q_1, \mathcal{D})$. However, the satisfiably extendable assignments in $\Gamma_M(Q_1, \mathcal{D})$ are

$$\{u \mapsto 1, n \mapsto \text{'John Doe'}, i \mapsto 1\}$$

$$\{u \mapsto 1, n \mapsto \text{'John Doe'}, i \mapsto 2\}$$

$$\{u \mapsto 2, n \mapsto \text{'Jane Roe'}, i \mapsto 1\}$$

whereas the satisfiably extendable assignments in $\Gamma_M(Q_2, \mathcal{D})$ are

$$\{n \mapsto \text{'John Doe'}\}$$

$$\{n \mapsto \text{'Jane Roe'}\}$$

The difference stems from the fact that u and i are multiset variables in Q_1 but are set variables in Q_2 .

We now define evaluation semantics for our query language. Our semantics are based on the ones proposed by Cohen. [23]

Definition 4.1.3 (Combined semantics). *Let Q be a query with head $Q(\bar{t})$, and let \mathcal{D} be a dataset. The result of applying Q to \mathcal{D} under combined semantics, is defined as*

$$Res(Q, \mathcal{D}) = \{ \{ Q(\gamma\bar{t}) \mid \gamma \in \Gamma_M(Q, \mathcal{D}) \} \}$$

We can generalize this to a set of queries \mathbf{Q} as follows:

$$Res(\mathbf{Q}, \mathcal{D}) = \bigsqcup_{Q \in \mathbf{Q}} Res(Q, \mathcal{D})$$

where \bigsqcup is used to denote multiset union.

In our running example $Res(Q_1, \mathcal{D})$ is the dataset

$$\{ \{ Q_1('John\ Doe'), Q_1('John\ Doe'), Q_1('Jane\ Roe') \} \}$$

whereas $Res(Q_2, \mathcal{D})$ is the dataset

$$\{ \{ Q_1('John\ Doe'), Q_1('Jane\ Roe') \} \}$$

In general, $Res(Q, \mathcal{D})$ will contain exactly one tuple for each element of $\Gamma_M(Q, \mathcal{D})$.

4.1.2 Equivalence, Homomorphisms and Foldings

Under what circumstances can we infer that two queries return the same answer on every possible dataset? We provide a criterion based on query homomorphisms [19, 23].

Definition 4.1.4 (Multiset-homomorphism). *Let $Q(\bar{t}) :- R, M$ and $Q'(\bar{t}') :- R', M'$ be queries, and let φ be a mapping from terms of Q to terms of Q' . We apply φ to atoms in the obvious way, i.e., $\varphi(A(\bar{t}; i)) = A(\varphi\bar{t}; \varphi i)$ for each atom $A(\bar{t}; i)$ in R . We say that φ is a **multiset-homomorphism** from Q to Q' if it satisfies the following conditions:*

- (i) $\varphi\bar{t} = \bar{t}'$.
- (ii) φ is the identity mapping on constants.
- (iii) $\varphi R \subseteq R'$.
- (iv) $\varphi M \subseteq M'$ and $\varphi y \neq \varphi y'$ for every two distinct variables $y, y' \in M$.

If Q and Q' only have set and distinguished variables (but not multiset variables) then the existence of a homomorphism $\varphi : Q \rightarrow Q'$ implies that $Res(Q, \mathcal{D}) \supseteq Res(Q', \mathcal{D})$ on any dataset \mathcal{D} . In other words, if $Res(Q', \mathcal{D})$ contains n copies of a tuple t then $Res(Q, \mathcal{D})$ must contain *at least* n copies. This well-known result is due to Chandra and Merlin.

[19] In general, the criterion is not guaranteed to hold for other types of queries. [21] However, it *does* hold if φ is bijective on multiset variables. The proof of the following Theorem is identical to that of Cohen's Theorem 3.4. [23]

Theorem 4.1.5. *Let $\varphi : Q \rightarrow Q'$ be a homomorphism that is a bijection from the multiset variables of Q onto those of Q' . Then $Res(Q, \mathcal{D}) \supseteq Res(Q', \mathcal{D})$ on any dataset \mathcal{D} .*

Two queries are said to be **multiset-homomorphic**, or **homomorphic** for short, denoted as $Q \equiv Q'$, if and only if there are multiset-homomorphisms from Q to Q' and from Q' to Q . Queries that are homomorphic return the same answer on every dataset.

Corollary 4.1.6. *If $Q \equiv Q'$ then $Res(Q, \mathcal{D}) = Res(Q', \mathcal{D})$ on every dataset \mathcal{D} .*

Proof. Let $\varphi : Q \rightarrow Q'$ and $\varphi' : Q' \rightarrow Q$ be homomorphisms. Since φ and φ' are injective on multiset variables, Q and Q' must have the same number of multiset variables, and therefore φ is a bijection on multiset variables. The previous Theorem implies that $Res(Q, \mathcal{D}) \supseteq Res(Q', \mathcal{D})$. An analogous argument in the other direction shows that $Res(Q, \mathcal{D}) \subseteq Res(Q', \mathcal{D})$, and therefore $Res(Q, \mathcal{D}) = Res(Q', \mathcal{D})$. \square

It is easy to verify that \equiv is an equivalence relation, so that the following hold:

- (i) **Reflexivity:** $Q \equiv Q$.
- (ii) **Symmetry:** If $Q \equiv Q'$ then $Q' \equiv Q$.
- (iii) **Transitivity:** If $Q \equiv Q'$ and $Q' \equiv Q''$ then $Q \equiv Q''$.

We now demonstrate an application of these results. Earlier we argued intuitively that

```
SELECT U1.name FROM User U1
```

returns the same answer as the query

```

SELECT U1.name FROM User U1
WHERE U1.uid IN (SELECT U2.uid FROM User U2)

```

on every possible dataset. We can prove that the corresponding conjunctive queries

$$Q_1(n) :- \text{User}(u, n; i_1), \{u, i_1\}$$

$$\text{and } Q_4(n_1) :- \text{User}(u, n_1; i_1) \wedge \text{User}(u, n_2; i_2), \{u, i_1\}$$

return the same answer on every dataset. Define $\varphi : Q_1 \rightarrow Q_4$ and $\varphi' : Q_4 \rightarrow Q_1$ by

$$\varphi = \{u \mapsto u, n \mapsto n_1, i_1 \mapsto i_1\}$$

$$\varphi' = \{u \mapsto u, n_1 \mapsto n, i_1 \mapsto i_1, n_2 \mapsto n, i_2 \mapsto i_1\}$$

It is straightforward to verify that φ and φ' are homomorphisms, and therefore equivalence follows from Corollary 4.1.6.

We now consider the complexity of checking whether Q and Q' are homomorphic. In general, the problem is NP-hard in the number of body atoms of Q and Q' . However, since real queries typically do not have many body atoms, the check can generally be performed quite fast in practice.

Proposition 4.1.7. *Given Q and Q' , the problem of determining whether $Q \equiv Q'$ is NP-complete w.r.t. the number of body atoms in Q and Q' .*

Proof. Given a homomorphism $\varphi : Q \rightarrow Q'$ (or $\varphi' : Q' \rightarrow Q$), we can check whether φ satisfies the properties from Definition 4.1.4 in polynomial time. Hence, the problem is in NP. Theorem 11 of Chandra and Merlin [19] tells us that the problem is NP-hard for conjunctive queries under set semantics. Since our language generalizes conjunctive queries under set semantics, it must be at least NP-hard for queries in our language. \square

A special case occurs when there is a homomorphism φ from Q to Q' whose inverse is also a homomorphism. In this case, we say that Q and Q' are **isomorphic**.

Definition 4.1.8. *Queries Q and Q' are **isomorphic** if there exists a bijective map φ from the terms in Q to the terms in Q' such φ and φ^{-1} are both homomorphisms.*

Our definition of query homomorphisms implies that isomorphic queries are identical up to reordering of body atoms and renaming of variables. Furthermore, if φ is an isomorphism then φ and φ^{-1} are homomorphisms, and therefore queries that are isomorphic are homomorphic as well.

We next define **foldings** for queries in our language; our results generalize classical results for conjunctive queries under set semantics. [19] Recall the queries

$$Q_1(n) :- \text{User}(u, n; i_1), \{u, i_1\}$$

$$\text{and } Q_4(n_1) :- \text{User}(u, n_1; i_1) \wedge \text{User}(u, n_2; i_2), \{u, i_1\}$$

from our running example. The two queries are homomorphic (and therefore return the same answer on every dataset), but Q_1 contains strictly fewer body atoms than Q_4 . In this case, Q_1 is said to be a *folding* of Q_4 .

Definition 4.1.9. *Let Q, Q' be conjunctive queries that have the same head variables. Q' is a **folding** of Q if (i) Q' contains a subset of the body atoms of Q , and (ii) $Q \equiv Q'$.*

Given a query Q , it is always possible to find a **minimal** folding of Q , i.e., a folding with a minimal number of body atoms. For example, Q_1 is a minimal folding of Q_4 . The next Theorem shows that the minimal folding is always unique up to isomorphism.

Theorem 4.1.10. *Let $Q \equiv Q'$. Let Q_F be a minimal folding of Q , and let Q'_F be a minimal folding of Q' . Then Q_F is isomorphic to Q'_F .*

Proof. Since \equiv is transitive, $Q_F \equiv Q \equiv Q' \equiv Q'_F$. Hence, there exist query homomorphisms $\varphi : Q_F \rightarrow Q'_F$ and $\varphi' : Q'_F \rightarrow Q_F$. This means that $\varphi' \circ \varphi$ maps every body atom

of Q_F to a body atom of Q_F . We claim that $\varphi' \circ \varphi$ is a bijection on the atoms of Q_F atoms (and therefore on the terms of Q_F as well). Suppose not. Then Q_F is homomorphic to $(\varphi' \circ \varphi)(Q_F)$, which has the same head as Q_F but strictly fewer body atoms. This contradicts our assumption that Q_F is minimal.

Since $(\varphi' \circ \varphi)$ is bijective, it induces a permutation of the terms of Q_F . Hence, there must be some $n \geq 1$ such that $(\varphi' \circ \varphi)^n(Q_F) = Q_F$. This means that φ is a bijection from the atoms (resp. terms) of Q_F to those of Q'_F , with inverse $\varphi^{-1} = (\varphi' \circ \varphi)^{n-1} \circ \varphi'$. The conditions of Definition 4.1.4 ensure that φ maps the head of Q_F to that of Q'_F , and maps each constant in Q_F to a constant in Q'_F . It maps constants to themselves, which means it must define a bijection between variables in Q_F and Q'_F . Furthermore, it maps multiset variables to multiset variables, which means that it also maps set variables to set variables. It follows that φ is an isomorphism, completing the proof. \square

A proof and accompanying algorithm developed by Gottlob and Fermüller [32] can be adapted to show that finding a minimal folding of a query Q is co-NP-complete w.r.t. the number of body atoms in Q . Fortunately, few real queries have more than a handful of body atoms, and so this can usually be done quickly in practice.

4.1.3 Equality Predicates

It is reasonable to ask whether the expressive power of our query language could be increased by adding built-in support for equality predicates of the form $(x = y)$. It turns out that such predicates are redundant: variable unification serves exactly the same purpose as equality predicates do. Consider a query that checks for users who are friends with themselves.

$$Q_5(u_1) :- \text{Friend}(u_1, u_2; i) \wedge (u_1 = u_2), \emptyset$$

This query can be written without equality predicates as follows:

$$Q_6(u) :- \text{Friend}(u, u), \emptyset$$

For this reason we can accept database queries containing explicit equality predicates but will remove such predicates before applying the algorithms discussed elsewhere in this Chapter (such as searching for homomorphisms between queries).

Formally, we can model equality predicates by adding to the database schema a binary relation $R_=_$ whose contents consist of the multiset $\{(c, c) \mid c \text{ is a constant}\}$. We refer to an atom over $R_=_$ as an *equality atom*. If we wish to be pedantic, the query Q_5 defined above should be written as follows:

$$Q_5(u_1) :- \text{Friend}(u_1, u_2; i_1) \wedge R_=(u_1, u_2; i_2), \emptyset$$

However, we will frequently abuse notation by abbreviating $R_=(u_1, u_2; i_2)$ as $(u_1 = u_2)$.

We begin by grouping the terms in Q into *equivalence classes*. If the atom $R_=(e_1, e_2; i)$ appears as a body atom in Q then e_1 and e_2 are in the same equivalence class. In the query Q_5 , the variables u_1 and u_2 belong to the same equivalence class. Equivalence classes are closed under transitivity, and that every variable in Q appears in exactly one equivalence class. If there is an equivalence class that contains two distinct constants then we say that Q is *unsatisfiable*; otherwise, we say Q is *satisfiable*. For example, the query

$$Q_7() :- \text{Friend}(u_1, u_2; i_1) \wedge (u_1 = 3) \wedge (u_1 = 4)$$

is unsatisfiable because the constants 3 and 4 appear in the same equivalence class. This makes sense: u_1 cannot be equal to both 3 and 4 at the same time.

Proposition 4.1.11. *If Q is not satisfiable then $\text{Res}(Q, \mathcal{D}) = \emptyset$ for every dataset \mathcal{D} .*

Proof. We will show the contrapositive: if $\text{Res}(Q, \mathcal{D})$ is nonempty for some dataset \mathcal{D} then Q is satisfiable. It suffices to show that if $\text{Res}(Q, \mathcal{D})$ is nonempty and c and

```

procedure REMOVEEQUALITY( $Q$ )
  if  $Q$  does not have any equality atoms then
    return  $Q$ 
  else
    Let  $Q'$  be a copy of  $Q$ 
    Let  $R_=(e_1, e_2; i)$  be an equality atom in  $Q'$ 
    Remove  $R_=(e_1, e_2; i)$  from the body of  $Q'$ 
     $e_3 \leftarrow \text{GETUNIFIER}(e_1, e_2)$ 
    Replace all occurrences of  $e_1$  and  $e_2$  in  $Q'$  with  $e_3$ 
    return  $Q'$ 
  end if
end procedure

procedure GETUNIFIER( $e_1, e_2$ )
  if  $e_1$  or  $e_2$  is a constant  $c$  then
    return  $c$ 
  else if  $e_1$  or  $e_2$  is distinguished then
    return a fresh distinguished variable
  else if  $e_1$  or  $e_2$  is multiset-existential then
    return a fresh multiset-existential variable
  else
    return a fresh set-existential variable
  end if
end procedure

```

Figure 4.3: Removing equality predicates through variable unification.

c' are constants in the same equivalence class then $c = c'$. Since c and c' are in the same equivalence class, there must be terms e_1, \dots, e_n such that Q contains the body atoms $R_=(e_1, e_2), \dots, R_=(e_{n-1}, e_n)$ and $c = e_i, c' = e_j$ for some $i, j \in \{1, 2, \dots, n\}$. This means that $R_=(e_1, e_2), \dots, R_=(e_{n-1}, e_n)$ are all satisfied, and therefore $e_1 = e_2 = \dots = e_n$; it follows that $c = c'$. \square

For the remaining cases, we rely on the REMOVEEQUALITY procedure in Figure 4.3. Each time REMOVEEQUALITY is called, it will either leave Q unchanged (if Q does not contain any equality atoms) or will generate a new query Q' with one less equality atom than Q . We can therefore apply REMOVEEQUALITY repeatedly until we are left with a query

that has no equality atoms. It remains to show that the resulting query is homomorphic to Q . Correctness of this procedure is based on the following Proposition.

Proposition 4.1.12. *Let Q be a satisfiable query, and let $Q' = \text{REMOVEEQUALITY}(Q)$. Then $\text{Res}(Q, \mathcal{D}) = \text{Res}(Q', \mathcal{D})$ for every dataset \mathcal{D} .*

Proof. When Q does not contain any equality atoms, $Q' = Q$, and the result follows immediately. Otherwise, fix a dataset \mathcal{D} and an element \bar{t}_c in the codomain of $\text{Res}(Q, \mathcal{D})$. It suffices to show that \bar{t}_c has the same multiplicity in $\text{Res}(Q', \mathcal{D})$. We will provide a bijection between elements $\gamma \in \Gamma_M(Q, \mathcal{D})$ which satisfy $\gamma(\bar{t}) = \bar{t}_c$ and elements $\gamma' \in \Gamma_M(Q', \mathcal{D})$ which satisfy $\gamma'(\bar{t}) = \bar{t}_c$.

Suppose we are given $\gamma \in \Gamma(Q, \mathcal{D})$; we must construct suitable $\gamma' \in \Gamma(Q', \mathcal{D})$. Let $R_=(e_1, e_2; i)$ be the unique atom of Q that does not appear in Q' . On terms in Q other than e_1, e_2 , and i , we set $\gamma'(t) = \gamma(t)$. We extend γ' by setting $\gamma'(e_3) = \gamma(e_1)$, so that $\gamma'(e_3) = \gamma(e_2)$ as well. Let A be a body atom of Q , and let A' be the corresponding body atom of Q' . Then $\gamma'(A') = \gamma(A)$ by construction, and therefore $\gamma' \in \Gamma(Q', \mathcal{D})$.

For the other direction, let $\gamma' \in \Gamma(Q', \mathcal{D})$. Then we can construct a corresponding $\gamma \in \Gamma(Q, \mathcal{D})$. We let γ coincide with γ' on all terms other than e_3 . We then extend γ by setting $\gamma(e_1) = \gamma(e_2) = \gamma'(e_3)$ and $i = 1$. (The fact that Q is satisfiable ensures that constants in γ and γ' are mapped to themselves.)

The argument above gives us a bijection between $\gamma \in \Gamma(Q, \mathcal{D})$ and $\gamma' \in \Gamma(Q', \mathcal{D})$. We claim that its restriction to distinguished and multiset-existential variables gives us a bijection between $\gamma_M \in \Gamma(Q, \mathcal{D})$ and $\gamma'_M \in \Gamma_M(Q', \mathcal{D})$. By the previous argument, $\gamma_M \in \Gamma_M(Q, \mathcal{D})$ is satisfiably extendable with $\gamma_M(\bar{t}) = \bar{t}_c$ if and only if $\gamma'_M \in \Gamma_M(Q', \mathcal{D})$ is satisfiably extendable with $\gamma'_M(\bar{t}) = \bar{t}_c$. We must now consider several cases. (Some cases are omitted because they are redundant; these cases can be handled by swapping t_1 and t_2

in the argument below.)

- If neither t_1 nor t_2 is multiset-existential then $\gamma'_M = \gamma_M$, and we're done.
- If t_1 is multiset-existential and t_2 is set-existential then our bijection sets $\gamma'_M(t_3) = \gamma_M(t_1)$, and keeps the remaining multiset and distinguished variables the same.
- If t_1 is multiset-existential and t_2 is distinguished or constant then t_3 is distinguished or constant as well. In this case, for any assignment γ defined on the distinguished variables other than t_1 there is exactly one possible assignment of $\gamma(t_1)$ that is satisfiably extendable and which satisfies $\gamma(\bar{t}) = \bar{t}_c$.
- If t_1 and t_2 are both multiset-existential then we reuse the previous bijection, which took assignments γ such that $\gamma(t_1) = \gamma(t_2)$ to assignments γ' such that $\gamma'(t_3) = \gamma(t_1)$. \square

4.2 Query Rewritings

Under what circumstances can we say that a set of views $\mathbf{V} = \{V_1, V_2, \dots, V_n\}$ reveals as much information as a query Q ? Formally, we would like to ensure that there is a deterministic function f that computes the answer to Q from the answers to the views in \mathbf{V} on every dataset \mathcal{D} . In this case, it is safe to say that the views in \mathbf{V} represent an upper bound on the information disclosed by Q : any information that could be learned by looking at the answer to Q could also be learned by looking at the views in \mathbf{V} .

Unfortunately, it is not known whether the problem of checking whether an appropriate f exists is tractable in general; in fact, we don't even know whether it is *decidable* in a setting that is closely related to the one considered here. [41] We impose two practical restrictions in order to ensure tractability. First, rather than allowing f to range over *arbitrary* functions we restrict our attention to functions f that can be expressed as con-

conjunctive queries under combined semantics. And second, rather than checking whether f and Q return the same answer on every possible dataset we perform a conservative check based on the existence of certain query homomorphisms. Under these restrictions, f is referred to as a *homomorphic rewriting of Q using \mathbf{V}* ; the existence of such a rewriting is sufficient (but not necessary) to ensure determinacy.

4.2.1 Rewritings and Expansions

Informally, a *rewriting Q* is a conjunctive query whose predicates are views in \mathbf{V} rather than base relations. The result of Q on a dataset \mathcal{D} , denoted $Res(Q, \mathcal{D})$, is obtained by (i) evaluating views in \mathbf{V} on \mathcal{D} to obtain a new dataset \mathcal{D}' and then (ii) evaluating Q on this new dataset \mathcal{D}' . This is formalized as follows.

Definition 4.2.1 (Evaluation of rewritings). *Let Q be a rewriting using \mathbf{V} . Then $Res(Q, \mathcal{D}) = Res(Q, \mathcal{D}')$ where $\mathcal{D}' = Res(\mathbf{V}, \mathcal{D})$.*

For example, consider the views

$$V_8(u, n) :- \text{User}(u, n; i_1) \wedge \text{Friend}(4, u; i_2), \{i_1\}$$

$$V_9(u_1, u_2) :- \text{Friend}(u_1, u_2; i), \{u_1, u_2, i\}$$

The first view lists the UIDs and names of User 4's friends, while the second lists all the friendships in the dataset. What follows is a rewriting using $\{V_8, V_9\}$:

$$Q_{10}(n, u_2) :- V_8(u_1, n; i_1) \wedge V_9(u_1, u_2; i_2), \{u_1, i_1, i_2\}$$

Suppose we wish to evaluate Q_{10} on the dataset from Figure 4.1. We first evaluate $\mathbf{V} = \{V_8, V_9\}$ on this dataset to obtain a new dataset:

$$\mathcal{D}' = \{ \{ V_8(2, \text{'Church, Alonzo'}), V_9(1, 3), V_9(3, 1), V_9(2, 4), V_9(4, 2) \} \}$$

We then evaluate Q_{10} on \mathcal{D}' to obtain the final answer:

$$Res(Q_{10}, \mathcal{D}') = \{ \{ Q_{10}('Church, Alonzo', 4) \} \}$$

An important property of our query language is that rewritings are closed under composition in the following sense: Suppose Q is a rewriting using \mathbf{V} . We can efficiently compute a conjunctive query Q_+ defined directly over \mathcal{D} such that Q and Q_+ return the same answer on every dataset; we refer to Q_+ as the *expansion of Q* . The expansion is computed as follows:

Definition 4.2.2 (Query Expansion). *Let*

$$Q(\bar{t}) :- A_1(\bar{t}_1; i_1) \wedge \dots \wedge A_m(\bar{t}_m; i_m), M$$

be a rewriting using $\mathbf{V} = \{V_1, V_2, \dots, V_n\}$. We define $Q_+(\bar{t})$ to be the query obtained by replacing each $A_k(\bar{t}_k; i_k)$ with the body of the corresponding query $V_k \in \mathbf{V}$. We replace each variable in the head of V_k with the corresponding variable in \bar{t}_k , and rename the remaining variables of V_k as needed to avoid capture. M' contains the regular multiset variables in M . It also contains the multiset variables of every body atom $A_k(\bar{t}_k; i_k)$ such that $i_k \in M$.

In our running example, Q_{10} has the following expansion:

$$Q_{10_+}(n, u_2) :- \text{User}(u_1, n; i_1) \wedge \text{Friend}(4, u_1; i_2) \wedge \text{Friend}(u_1, u_2; i_3), \{u_1, i_1, i_3\}$$

The first two body atoms of Q_{10_+} come from V_8 , while the third comes from V_9 . We next verify the correctness of the expansion process described above. Formally, we must show that Q yields the same answer as Q_+ on every dataset \mathcal{D} .

Theorem 4.2.3. *$Res(Q, \mathcal{D}) = Res(Q_+, \mathcal{D})$ for every dataset \mathcal{D} .*

Proof. Fix a dataset \mathcal{D} . We will provide a bijection between elements of $\gamma \in \Gamma_M(Q, \mathcal{D})$ which satisfy $\gamma(\bar{t}) = \bar{t}_c$ and elements of $\gamma_+ \in \Gamma_M(Q_+, \mathcal{D})$ which satisfy $\gamma_+(\bar{t}) = \bar{t}_c$. The key insight is that for each atom $V_k(\bar{t}_k; i_k)$ we can fix a bijection $\beta_k(\gamma_k)$ between assignments γ_k of the multiset variables in the view V_k and assignments of the copy variable i_k .

Choose any $\gamma_+ \in \Gamma_M(Q_+, \mathcal{D})$ that is satisfiably extendable to $\gamma'_+ \in \Gamma(Q_+, \mathcal{D})$. We will construct a corresponding assignment $\gamma' \in \Gamma(Q, \mathcal{D})$. First, let γ' coincide with γ'_+ on all the ordinary variables of Q . For the remaining variables of Q (which are all copy variables) let $\gamma'(i_k) = \beta_k(\gamma'_k)$, where γ'_k is the restriction of γ'_+ to multiset variables in V_k .

Now γ' is defined on all the variables in Q . Furthermore, $\gamma' \in Res(Q, \mathcal{D})$ because of the way we defined Q_+ . Moreover, $\gamma'(\bar{t}) = \bar{t}_c$ because γ' and γ'_+ coincide on all the ordinary variables of Q , including those in \bar{t} . Define γ to be the restriction of γ' to multiset variables of Q , and notice that γ_+ is the restriction of γ'_+ to multiset variables of Q_+ . This gives us a correspondence between elements of $\gamma \in \Gamma_M(Q, \mathcal{D})$ which satisfy $\gamma(\bar{t}) = \bar{t}_c$ and elements of $\gamma_+ \in \Gamma_M(Q_+, \mathcal{D})$ which satisfy $\gamma_+(\bar{t}) = \bar{t}_c$. In fact, the correspondence is a bijection. The reason is that γ and γ_+ coincide on all multiset variables common to Q and Q_+ , and the cross product $\beta_1 \times \dots \times \beta_m$ induces a bijection between the remaining multiset variables of Q and Q_+ respectively. \square

We now provide a criterion for checking whether the answers to a set of views determine the answer to a given query. As discussed above, our criterion is a conservative approximation to view determinacy. It generalizes a well-known criterion for conjunctive queries under set semantics; for details, we refer the interested reader to the survey of Levy et al. [38]

Definition 4.2.4 (Homomorphic rewriting). *Let Q be a query and let \mathbf{V} be a set of views. We say a query Q' is a homomorphic rewriting of Q using \mathbf{V} if (i) Q' is a rewriting using \mathbf{V} , and (ii) Q'_+ is homomorphic to Q .*

If Q' is a homomorphic rewriting of Q using \mathbf{V} then the answer to Q is uniquely determined by the answers to the views in \mathbf{V} on any possible dataset, and therefore \mathbf{V} represents an upper bound on the information needed to answer Q . In the example above, Q_{10} is a rewriting of Q_{10+} using $\{V_8, V_9\}$. Hence, $\{V_8, V_9\}$ represents an upper bound on the information needed to answer Q_{10+} .

4.2.2 Finding Rewritings

In the previous section, we showed that the existence of a rewriting implies determinacy, and proposed the use of rewritings as a conservative approximation to determinacy. However, rewritings are only useful if we can efficiently check whether the answers to a set of queries determines the answer to a given view. In this section we show that performing this check is an NP-complete problem. Since the complexity depends only on the sizes of the query and view definitions and not on the underlying database, this is essentially a positive result – at least when the number of views is not too large.

Our proof is based on the following property: if a query Q has a rewriting using \mathbf{V} then it must have a rewriting that is *small* in the sense that it does not have too many body atoms. This makes it feasible (although not necessarily efficient) to perform a brute-force search over the space of possible rewritings.

Theorem 4.2.5. *Let Q be a query with n body atoms, and let \mathbf{V} be a set of arbitrary (possibly multi-atom) views. The following are equivalent:*

- (i) *There exists a rewriting of Q using \mathbf{V} .*
- (ii) *There exists a rewriting of Q using \mathbf{V} that contains at most n body atoms.*

Proof.

(ii) implies (i): Trivial.

(i) implies (ii): Suppose (i) holds. Then there must exist a rewriting Q^R of Q using \mathbf{V} . Let Q_+^R denote the expansion of Q^R . By definition, there must exist homomorphisms $\varphi : Q \rightarrow Q_+^R$ and $\varphi^R : Q_+^R \rightarrow Q$. In particular, φ must map each body atom B_i of Q onto some body atom A_i of Q_+^R . Let Q^S be a query with the same head as Q^R whose body consists of the conjunction of every atom in Q^R that is associated with some A_i . This query is well-formed: our choice of φ ensures that every distinguished variable in the body of Q^R also appears in the body of Q^S . Moreover, φ maps each body atom of Q to a body atom of Q_+^S , and φ^R maps each body atom of Q_+^S to a body atom of Q . It follows that Q and Q_+^S are homomorphic, and therefore Q^S is a rewriting of Q using \mathbf{V} . Furthermore, Q^S contains no more body atoms than Q and therefore satisfies condition (ii). \square

With the help of the preceding Theorem, we now show that determining whether a query Q has a rewriting using a set of views \mathbf{V} is an NP-complete problem.

Corollary 4.2.6. *Given a query Q and a set of views \mathbf{V} , the problem of determining whether there exists a homomorphic rewriting of Q using \mathbf{V} is NP-complete.*

Proof. Results from previous work (e.g., Theorem 3.7 of Levy et al. [38]) demonstrate that the problem is NP-hard even when we restrict our attention to conjunctive queries under set semantics. Given a rewriting Q^R of Q using \mathbf{V} and homomorphisms $\varphi : Q \rightarrow Q_+^R$ and $\varphi^R : Q_+^R \rightarrow Q$, we can verify in polynomial time that (i) Q_+^R is the correct expansion of Q and (ii) the homomorphisms φ and φ^R are valid. Hence, the problem is in NP. \square

The problem is much easier when Q and each view in \mathbf{V} contains exactly one body atom. In this case, we can check for rewritings in linear time.

Proposition 4.2.7. *Let Q be a single-atom query, and let \mathbf{V} be a set of single-atom views. We can determine in linear time whether there exists a rewriting of Q using \mathbf{V} .*

Proof. From Theorem 4.2.5 above, we know that if there is a rewriting of Q using \mathbf{V} then there must be a rewriting that contains exactly one body atom. If Q^R is such a rewriting then Q_+^R must be isomorphic to Q because both Q and Q_+^R are single-atom conjunctive queries and are therefore minimal folding.

Consequently, it suffices to check whether there exists a single-atom rewriting Q^R of Q using \mathbf{V} . If such a rewriting exists, the unique body atom of Q^R must reference some view $V \in \mathbf{V}$. Observe that Q_+^R and Q must be isomorphic even when we restrict our attention to variables that are distinguished in V , and there is (up to isomorphism) only one choice of Q^R that has a chance of satisfying this criterion. And since Q and Q_+^R are single-atom queries, we can check whether there are valid homomorphisms from Q to Q_+^R and from Q_+^R to Q in linear time.

Hence, we can check for a rewriting of Q using $\{V\}$ in linear time. We can repeat this check for each $V \in \mathbf{V}$ to determine whether Q has a rewriting using \mathbf{V} . \square

4.3 Disclosure Labeling under Combined Semantics

Earlier in this section we defined rewritings for queries under combined semantics, and discussed the complexity of rewriting queries using views. We now integrate these results with the theory of Disclosure Labeling introduced in Chapter 3 in order to provide a formal framework for reasoning about the information needed to answer queries under combined semantics. The results in this section generalize previous results for conjunctive queries under set semantics from Chapter 3.

4.3.1 An Order Based on Equivalent Rewritings

We begin by reviewing some basic definitions from Chapter 3. We assume that all the views under consideration are drawn from some finite universe U . Recall that a *disclosure order* is a binary relation on sets of views which satisfies three properties:

- (i) If $V_1 \subseteq V_2$ then $V_1 \preceq V_2$.
- (ii) If $V_1 \preceq V_2$ and $V_2 \preceq V_3$ then $V_1 \preceq V_3$.
- (iii) If $V_1 \preceq V_3$ and $V_2 \preceq V_3$ then $V_1 \cup V_2 \preceq V_3$.

Intuitively, $V_1 \preceq V_2$ means that V_2 reveals more information about the dataset than V_1 . The first axiom says that adding new elements to a set of views can only increase the amount of information that it reveals about the dataset. The second is a standard transitivity condition. The third ensures that upper bounds on information disclosure remain meaningful even when information is combined from multiple views.

We might be tempted to say that $V_1 \preceq V_2$ if and only if the answers to V_2 uniquely determine the answers to V_1 on any possible dataset. However, we have already argued that this interpretation – although attractive in theory – is computationally infeasible. We are not aware of any algorithms that can perform this check efficiently, and it is an open question whether a closely related formulation of the problem is even *decidable*. [41]

Instead, we adopt a conservative approximation to determinacy that is based on query rewritings. Under this interpretation, $V \preceq V'$ if every $V \in V$ has an equivalent rewriting using V' . The definition comes with an intuitive interpretation: $V \preceq V'$ if and only if we can prove that the answers to the queries in V' uniquely determine the answers to the queries in V on every possible dataset. It is straightforward to show that this defines a disclosure order:

Proposition 4.3.1. *The binary relation defined above, which is based on query rewritings, is a disclosure order.*

Proof. We verify each of the axioms of a disclosure order.

- (i) If $\mathbf{V} \subseteq \mathbf{V}'$ then $\mathbf{V} \leq \mathbf{V}'$: This follows immediately from the fact that every element of \mathbf{V} can be rewritten in terms of itself.
- (ii) If $\mathbf{V} \leq \mathbf{V}''$ and $\mathbf{V}' \leq \mathbf{V}''$ then $\mathbf{V} \cup \mathbf{V}' \leq \mathbf{V}''$: If $V \in \mathbf{V} \cup \mathbf{V}'$ then $V \in \mathbf{V}$ or $V \in \mathbf{V}'$; assume without loss of generality that $V \in \mathbf{V}$. By assumption, $\mathbf{V} \leq \mathbf{V}''$, and therefore V has an equivalent rewriting using \mathbf{V}'' .
- (iii) If $\mathbf{V} \leq \mathbf{V}'$ and $\mathbf{V}' \leq \mathbf{V}''$ then $\mathbf{V} \leq \mathbf{V}''$: Each element of \mathbf{V} has an equivalent rewriting using \mathbf{V}' . We replace each body atom in this rewriting with the rewriting of the corresponding element of \mathbf{V}' using \mathbf{V}'' using Theorem 4.2.3 above. \square

In Section 3.3.2 we provided efficient algorithms for reasoning about information disclosure under the assumption that sets of views were *decomposable*. Intuitively, this means that if the answer to a query Q is determined by a set of views then it must be determined by one of the views on its own.

Definition 4.3.2 (Decomposability). *Decomposable holds if for every set of views $\mathbf{V}_1, \mathbf{V}_2$ and every V such that $\{V\} \leq \mathbf{V}_1 \cup \mathbf{V}_2$ we have either $\{V\} \leq \mathbf{V}_1$ or $\{V\} \leq \mathbf{V}_2$.*

Although this property does not hold for conjunctive queries in general, it *does* hold in the special case where Q is a single-atom query. This result for queries under combined semantics generalizes earlier results for queries under set semantics from Section 3.4.

Proposition 4.3.3. *Suppose that Q is a single-atom query and let \mathbf{V} and \mathbf{V}' be sets of (arbitrary) views under combined semantics. If $\{Q\} \leq \mathbf{V} \cup \mathbf{V}'$ then $\{Q\} \leq \mathbf{V}$ or $\{Q\} \leq \mathbf{V}'$.*

Proof. Suppose that $\{Q\} \leq \mathbf{V} \cup \mathbf{V}'$. By Theorem 4.2.5, there exists a rewriting Q^S of Q using $\mathbf{V} \cup \mathbf{V}'$ such that Q^S contains exactly one body atom and $Q^S_+ \equiv Q$. The unique body atom of Q^S must originate from either \mathbf{V} or \mathbf{V}' . If it originates from \mathbf{V} then $Q^S \leq \mathbf{V}$, as desired. Analogously, if it originates from \mathbf{V}' then $Q^S \leq \mathbf{V}'$. \square

4.3.2 Labeling Single-Atom Queries under Combined Semantics

We next review the theory of disclosure labeling from Section 3.2, which allows us to represent the information needed to answer a query Q in terms of the information revealed by a set of security views \mathbf{V} . The security views are defined by a human administrator and reveal known types of information about the dataset.

Formally, a disclosure labeler ℓ relates the information revealed by an unknown set of queries to a set of *fixpoints* \mathcal{F} . For our purposes, \mathcal{F} consists of all possible subsets of \mathbf{V} , together with a maximal element \top that reveals all the information in the entire dataset.

Definition 4.3.4. A *disclosure labeler* is a map $\ell : \wp(\mathbf{U}) \rightarrow \wp(\mathbf{U})$ such that

- (i) If $\mathbf{V}_1 \subseteq \mathbf{U}$ then $\ell(\mathbf{V}_1) \equiv \mathbf{V}_2$ for some $\mathbf{V}_2 \in \mathcal{F}$.
- (ii) If $\mathbf{V} \in \mathcal{F}$ then $\ell(\mathbf{V}) \equiv \mathbf{V}$.
- (iii) If $\mathbf{V} \subseteq \mathbf{U}$ then $\mathbf{V} \leq \ell(\mathbf{V})$.
- (iv) If $\mathbf{V}_1, \mathbf{V}_2 \subseteq \mathbf{U}$ and $\mathbf{V}_1 \leq \mathbf{V}_2$ then $\ell(\mathbf{V}_1) \leq \ell(\mathbf{V}_2)$.

For single-atom queries and views, the algorithm defined in Section 3.3.2 provides us with a way to compute disclosure labels efficiently. This observation generalizes the results for conjunctive queries under set semantics from Section 3.4.1.

The results mentioned above rely on two black-box algorithms. The first computes the information that is *common* to two sets of views, while the second computes the

combined information from two sets of views. Under set semantics, it is possible to represent the information that is common to two single-atom conjunctive as another single-atom conjunctive query. For instance, define

$$V_{11}(u_1) :- \text{Friend}(u_1, u_2; i), \emptyset$$

$$V_{12}(u_2) :- \text{Friend}(u_1, u_2; i), \emptyset$$

Then the view

$$V_{13}() :- \text{Friend}(u_1, u_2; i), \emptyset$$

represents the information common to V_{11} and V_{12} , in the sense that, for any set of queries \mathbf{Q} , we have $\mathbf{Q} \leq \{V_{13}\}$ if and only if $\mathbf{Q} \leq \{V_{11}\}$ and $\mathbf{Q} \leq \{V_{12}\}$. In fact, Section 3.4.1 contains a general procedure for computing the information common to two single-atom views under set semantics. Computing the *combined* information from two sets of views is even easier: we simply take the union of the two sets.

Unfortunately, we encounter a combinatorial explosion when we try to generalize these results to conjunctive queries under combined semantics. For instance, to represent the information common to the views

$$V_{14}(u_1) :- \text{Friend}(u_1, u_2; i), \{u_2\}$$

$$V_{15}(u_2) :- \text{Friend}(u_1, u_2; i), \{u_1\}$$

we would need four views, defined as follows:

$$V_{16}() :- \text{Friend}(u_1, u_2; i), \{u_1, u_2\}$$

$$V_{17}() :- \text{Friend}(u_1, u_2; i), \{u_1\}$$

$$V_{18}() :- \text{Friend}(u_1, u_2; i), \{u_2\}$$

$$V_{13}() :- \text{Friend}(u_1, u_2; i), \emptyset$$

V_{13} is redundant, as its answer can be computed by removing duplicates from the answer to any of the other three views. However, the remaining three views are all mutually

incomparable. V_{16} reveals the number of distinct pairs (u_1, u_2) that appear in `Friend`, V_{17} reveals the number of distinct values of u_1 that appear in `Friend`, and V_{18} reveals the number of distinct values of u_2 . In general, we may need to construct a set with $2^n - 1$ distinct elements in order to represent the information that is common to n distinct single-atom views.

We address this problem by representing disclosure labels *symbolically* in disjunctive normal form. Instead of evaluating the GLB above, we store the unevaluated expression $(\Downarrow V_{14}) \sqcap (\Downarrow V_{15})$, which represents the information common to V_{14} and V_{15} using the notation introduced in Section 3.2.2. The symbolic representation is preferred for two reasons. First, it is easy to compute disclosure labels in this form using the algorithm derived in Section 3.3.2. And second, all of the relevant lattice-theoretic operations can be performed in polynomial time. More precisely, given inputs I_1 and I_2 whose representations have size m and n respectively, $I_1 \sqcup I_2$ can be computed in $O(m + n)$ time, $I_1 \sqcap I_2$ can be computed in $O(m \cdot n)$ time, and $I_1 \leq I_2$ and $I_1 = I_2$ can be checked with $O(m \cdot n)$ comparisons between pairs of single-atom views. Additional optimizations are discussed in Section 3.5.

4.3.3 Query Dissection

The analysis above permits us to efficiently compute a disclosure label for Q with respect to \mathbf{V} whenever Q and each $V \in \mathbf{V}$ contains exactly one body atom. In practice, however, relational queries frequently contain multiple body atoms, and so this result is of limited use. In this section, we discuss a disclosure labeler that maps a multi-atom query Q under combined semantics to a set of single-atom queries \mathbf{Q}' . The queries in \mathbf{Q}' contain enough information to determine the answer to Q on any possible dataset. Since the composition

of two disclosure labelers is itself a disclosure labeler, this map can be composed with the algorithm for single-atom conjunctive queries discussed in Section 4.3.2 to define a disclosure labeler that handles *arbitrary* conjunctive queries but only single-atom security views. This algorithm generalizes one for single-atom conjunctive queries under set semantics from Section 3.4.2.

The transformation from a multi-atom query Q to a set of single-atom queries \mathbf{Q}' is handled by a procedure called $\text{DISSECT}(Q)$ which operates as follows:

- (i) Compute a minimal folding Q_F of Q .
- (ii) Replace each set or multiset variable that appears in two or more distinct body atoms of Q with a fresh distinguished variable.
- (iii) For each body atom A_k of Q_F we define a query Q_k whose body consists of the atom A_k and whose head is obtained by restricting the head of Q_F to variables that appear in A_k .
- (iv) The algorithm's output contains a single-atom Q_k for each body atom A_k in Q_F .

For sets of queries, the algorithm can be generalized by dissecting each query individually and taking the union of all the resulting sets of single-atom queries. Formally,

$$\text{DISSECTALL}(\mathbf{V}) = \bigcup \{\text{DISSECT}(Q) : Q \in \mathbf{V}\}$$

It remains for us to show that DissectAll induces a disclosure labeler. In order to prove this fact, we need to prove two basic properties. The first guarantees that the output of $\text{DISSECTALL}(\mathbf{V})$ reveals at least as much information about the dataset as the input. The second guarantees that the output of DISSECTALL does not disclose any more information about the dataset than necessary.

Lemma 4.3.5. *The following properties hold of DISSECTALL :*

- (i) $\mathbf{Q} \leq \text{DISSECTALL}(\mathbf{Q})$ for any set of queries \mathbf{Q} .
- (ii) Suppose that \mathbf{Q} is an arbitrary set of queries and \mathbf{V} is a set of single-atom views. Then $\mathbf{Q} \leq \mathbf{V}$ implies $\text{DISSECTALL}(\mathbf{Q}) \leq \mathbf{V}$.

Proof. We prove each property in turn:

- (i) First suppose that \mathbf{Q} consists of a single query Q . In this case, it suffices to show that $\{Q\} \leq \text{DISSECT}(Q)$. We will show that each step of the algorithm above can only increase information disclosure. In Step (i), $Q_F \equiv Q$ by construction. In Step (ii), replacing existential variables with distinguished variables can only increase information disclosure. We can perform a multi-way join on the set of queries produced by Steps (iii) and (iv) to reconstruct the query generated in Step (ii).

We now generalize the result to sets of queries:

$$\mathbf{V} = \bigcup \{Q : Q \in \mathbf{V}\} \leq \bigcup \{\text{DISSECT}(Q) : Q \in \mathbf{V}\} = \text{DISSECTALL}(\mathbf{V})$$

- (ii) First suppose that \mathbf{Q} consists of a single query Q and that $\{Q\} \leq \mathbf{V}$. In this case, it suffices to show that $\text{DISSECT}(Q) \leq \mathbf{V}$. Let Q' be an equivalent rewriting of Q using \mathbf{V} , and let Q'_+ denote the expansion of Q' . Let Q_F be the folding of Q obtained in Step (i) of the DISSECT procedure. Then there is an isomorphism between Q_F and a subset of the body atoms of Q'_+ . Every distinguished variable that appears in an atom of Q_F must also be distinguished in the respective atom of Q'_+ . Moreover, every variable that appears in two distinct atoms of Q_F must also appear in two distinct atoms of Q'_+ , and therefore must be bound to distinguished variables in the corresponding views in \mathbf{V} . Putting these observations together, we conclude that for each atom $A \in \text{DISSECT}(Q)$ there must be a single-atom view $V \in \mathbf{V}$ such that $\{A\} \leq \{V\} \leq \mathbf{V}$. Taking the union over all such atoms, we conclude that $\text{DISSECT}(Q) \leq \mathbf{V}$, as desired.

We now prove the claim in its full generality. Suppose $\mathbf{Q} \leq \mathbf{V}$. Then $\{Q\} \leq \mathbf{V}$ for each $Q \in \mathbf{Q}$, and therefore $\text{DISSECT}(Q) \leq \mathbf{V}$ by the previous argument. Hence,

$$\text{DISSECTALL}(\mathbf{Q}) = \bigcup \{\text{DISSECT}(Q) : Q \in \mathbf{Q}\} \leq \mathbf{V} \quad \square$$

We are now ready to formally state and prove that DISSECTALL induces a disclosure labeler. Let \mathbf{U}_{single} denote the set of all single-atom views under combined semantics, and let \mathbf{U}_{conj} denote the set of *arbitrary* views under combined semantics. Then we will show that DISSECT is a disclosure labeler that maps a set of views in \mathbf{U}_{conj} to a set of views in \mathbf{U}_{single} .

Theorem 4.3.6. *DISSECTALL is a disclosure labeler.*

Proof. It now suffices to verify each of the disclosure labeler axioms in turn. Unless otherwise stated, \mathbf{Q} and \mathbf{Q}' are assumed to be arbitrary sets of conjunctive queries.

(i) $\mathbf{Q} \leq \text{DISSECTALL}(\mathbf{Q})$.

This is an immediate consequence of Part (i) of the preceding Lemma.

(ii) $\mathbf{Q} \leq \mathbf{Q}'$ implies $\text{DISSECTALL}(\mathbf{Q}) \leq \text{DISSECTALL}(\mathbf{Q}')$.

It suffices to show that $\mathbf{Q} \leq \mathbf{Q}'$ implies $\text{DISSECTALL}(\mathbf{Q}) \leq \text{DISSECTALL}(\mathbf{Q}')$. Since $\mathbf{Q} \leq \mathbf{Q}' \leq \text{DISSECTALL}(\mathbf{Q}')$ and $\text{DISSECTALL}(\mathbf{Q}')$ is a set of single-atom queries, this follows immediately from Part (ii) of the preceding Lemma.

(iii) If \mathbf{Q} is a set of single-atom queries then $\text{DISSECTALL}(\mathbf{Q}) \equiv \mathbf{Q}$.

It suffices to show that if Q is a single-atom query then $\text{DISSECT}(Q) = \{Q\}$. In Step (i) of the DISSECT algorithm, any single-atom query Q is a minimal folding, and so $Q_F = Q$. In Step (ii), no variable can appear in two distinct atoms of Q (because Q just contains one body atom), and so the query is unchanged. In Steps (iii) and (iv),

we will generate a single query whose head is the same as that of Q and whose unique body atom is the same as the unique body atom of Q .

(iv) If $\mathbf{Q} \leq \mathbf{Q}'$ then $\text{DISSECTALL}(\mathbf{Q}) \leq \text{DISSECTALL}(\mathbf{Q}')$.

The preceding Lemma implies that $\mathbf{Q} \leq \mathbf{Q}' \leq \text{DISSECTALL}(\mathbf{Q}')$, and therefore $\text{DISSECTALL}(\mathbf{Q}) \leq \mathbf{Q}'$ as well. \square

The take-away for this theorem is that by composing DISSECTALL with the single-atom disclosure labeling algorithm from the previous section, we can efficiently compute the disclosure labels for *arbitrary* conjunctive queries with respect to collections of single-atom security views. The result demonstrates that practical disclosure labeling algorithms exist for conjunctive queries under set, bag-set, and bag semantics, as well as under more realistic query languages that combine aspects of all three.

CHAPTER 5

TOWARDS PRACTICAL DISCLOSURE LABELING

In Chapter 3 we introduced a formal foundation for reasoning about information disclosure that was applicable to *any* query language – at least in principle. The basic idea was to express the information needed to answer an unknown query in terms of the information revealed by an administrator-defined set of *security views* that disclosed known types of information about the dataset. In Chapter 4, we extended this algorithm to a richer language that generalized conjunctive queries under bag, bag-set, and set semantics. Even in this case, however, we provided practical algorithms for reasoning about information disclosure only for a very restricted query language and an even more restricted class of security views. In this Chapter we show how the same principles can be applied to reason about information disclosure for a substantial fragment of the SQL query language and a class of security views that is rich enough to capture a diverse set of real-world security constraints.

The fragment of SQL that we deal with – which supports NULL values, negation, correlated subqueries, and disjunction – is rich enough that we do not try to reason about it directly. Instead, we translate SQL queries into an intermediate representation called *filter-project queries*, and then reason about the information disclosed by those intermediate filter-project queries. Filter-project queries are intended to hit a sweet spot between tractability and expressivity – they are rich enough to express a diverse set of security constraints, but are still restricted enough that we can reason about them efficiently and with strong theoretical guarantees.

The translation from SQL to filter-project queries is *not* lossless. Filter-project queries are defined in a restricted language that supports selections, projections, and semijoins, but not NULL values, negation, or disjunction. We guarantee that the filter-project queries

User			Friend	
UID	Name	Hobby	UID1	UID2
1	Babbage, Charles	math	1	3
2	Church, Alonzo	math	3	1
3	Lovelace, Ada	music	2	4
4	Turing, Alan	chess	4	2
5	von Neumann, John	history	4	5
			5	4

Figure 5.1: Sample database schema.

we generate contain enough information to answer the original SQL queries on *any* possible dataset. Consequently, we can *overestimate* the information needed to answer a SQL query, but will never *underestimate* the information that is needed.

The remainder of this Chapter is organized as follows. In Section 5.1 we define the language of *filter-project queries* and identify theoretical properties that allow us to efficiently reason about queries in this language. In Section 5.2 we define formal semantics for a fragment of SQL that includes correlated subqueries, negation, disjunction, and NULL values. In Section 5.3 we describe and prove the correctness of a procedure for translating queries in this fragment of SQL into filter-project queries. Finally, in Section 5.4 we provide an alternate formulation of the translation algorithm and informally discuss how it can be generalized to a much broader class of SQL queries.

We employ a running example inspired by the Facebook Query Language, or FQL, a SQL-like query language that can be used to access user data in the Facebook Apps developer API. [4] Our example schema (shown in Figure 5.1) has two relations: a `User` relation that stores a User ID (UID), a name, and a hobby for each user, and a `Friend` relation that keeps track of friendship relations between users.

5.1 Filter-Project Queries

We now define the language of *filter-project queries*, which combine aspects of conjunctive queries under bag and set semantics. Every filter-project query has a *head* that represents the query’s output, a *body* atom that is evaluated under bag semantics, and zero or more *filter* atoms that are evaluated under set semantics.

Consider the schema shown in Figure 5.1. We define two filter-project views. The first view reveals the UIDs and names of all the users in the database, while the second view reveals information about User 4’s friends.

$$V_1(u, n) :- \text{User}(u, n, h)$$

$$V_2(u, n) :- \text{User}(u, n, h) \times \text{Friend}(4, u)$$

This example illustrates a crucial point: in Facebook’s security model, it is important to distinguish between a query that reveals information about arbitrary users (such as V_1) and a query that only reveals information about friends of a particular user (such as V_2). This is not an isolated occurrence – normalization provides a natural way to represent access control policies. Access to information stored in one relation (`User`) is controlled by the contents of a different relation (`Friend`). Access control policies can then be specified using semijoins, as in the example above.

With this intuition in place, we are ready to formalize the language of filter-project queries. We use x, y, z to denote variables, and c, d to denote constants. A term, denoted as s, t , is either a variable or a constant. An *atom* takes the form $A(s_1, \dots, s_n)$ where A is a predicate of arity n . We will write $A(\bar{s})$ as shorthand for $A(s_1, \dots, s_n)$. A dataset \mathcal{D} is a multiset containing atoms of the form $A(c_1, \dots, c_n)$ where each A is a predicate of arity n and each c_i is a constant.

Definition 5.1.1. A filter-project query is a non-recursive expression of the form

$$Q(\bar{t}) :- A_1(\bar{t}_1) \times A_2(\bar{t}_2) \wedge \dots \wedge A_n(\bar{t}_n)$$

where each $A_i(\bar{t}_i)$ is an atom, and the following conditions hold:

- (i) Every variable in \bar{t} appears in \bar{t}_1 , and
- (ii) The existential variables in \bar{t}_1 are disjoint from those in $\bar{t}_2, \dots, \bar{t}_n$.

We refer to $Q(\bar{t})$ as the query *head*, $A_1(\bar{t})$ as the query *body*, and $A_2(\bar{t}_2), \dots, A_n(\bar{t}_n)$ as the query's *filter*. Variables that appear in the head are *distinguished*, while the remaining variables are *existential*.

Definition 5.1.1 imposes two restrictions on filter-project queries. These restrictions are not necessary to ensure the correctness of the queries' evaluation semantics, but will prove very useful later in this section when we develop algorithms for reasoning about information disclosure in filter-project queries. Condition (i) ensures that only variables that appear in the query's body can appear in the query head. For instance,

$$Q_3(u_2) :- \text{User}(u_1, n, h) \times \text{Friend}(u_1, u_2)$$

is *not* a valid filter-project query because the distinguished variable u_2 does not appear in the query body. Condition (ii) ensures that a query's body atom interacts through its filter atoms only through distinguished variables. For instance,

$$Q_4() :- \text{User}(u_1, n, h) \times \text{Friend}(u_1, u_2)$$

is *not* a valid filter-project query because the existential variable u_1 appears in both the body and the filter of the query. On the other hand,

$$Q_5(u_1) :- \text{User}(u_1, n, h) \times \text{Friend}(u_1, u_2)$$

is a valid filter-project query because u_1 is distinguished.

We do not define evaluation semantics for filter-project queries from scratch. Instead, we provide a translation of filter-project queries into the language of conjunctive queries under combined semantics defined in Chapter 4. The head of the conjunctive query is the same as the head of the filter-project query. We add a fresh copy variable to each of the query's body atoms. Existential variables that appear in the query's body atom become multiset-existential, while existential variables in the query's filter atoms become set-existential. The view V_1 and V_2 defined above are translated as follows:

$$V_6(u, n) :- \text{User}(u, n, h; i_1), \{h, i_1\}$$

$$V_7(u, n, h) :- \text{User}(u, n, h; i_1) \wedge F(4, u), \{i_1\}$$

We write $\text{BODYCOND}(Q)$ to refer to the result of this transformation on the query Q . In the example above, $V_6 = \text{BODYCOND}(V_1)$ and $V_7 = \text{BODYCOND}(V_2)$. Evaluation of a query Q is performed by (i) computing $\text{BODYCOND}(Q)$ and (ii) evaluating the resulting conjunctive query on the current dataset.

Performing this translation allows us to reuse many of the results derived in Chapter 4. For instance, two filter-project queries are said to be *homomorphic* if their corresponding conjunctive representations are *homomorphic*. For instance, the view

$$V_8(u, n) :- \text{User}(u, n, h) \bowtie \text{User}(u', n', h')$$

has the conjunctive query representation

$$V_9(u, n) :- \text{User}(u, n, h; i_1) \wedge \text{User}(u', n', h'; i_2), \{h, i_1\}$$

It is straightforward to show that V_6 is homomorphic to V_9 ; this means that the corresponding filter-project queries V_1 and V_8 return the same answer on every dataset.

It will be useful to define another function $\text{BODY}(Q)$ that denotes a translation which is the same as $\text{BODYCOND}(Q)$ on the head and body of Q but ignores any filter atoms in Q . In the example above, $\text{BODY}(V_1)$ is the same as $\text{BODY}(V_1)$, and $\text{BODYCOND}(V_2)$ is

$$V_{10}(u, n) :- \text{User}(u, n, h; i_1), \{i_1\}$$

5.1.1 Rewritings of Conjunctive Queries

We next define *rewritings* for filter-project queries. Traditionally, a *rewriting using \mathbf{V}* is a query whose body atoms reference views in \mathbf{V} instead of referencing base relations. The intuition is that a rewriting can be evaluated using *only* information revealed by the views in \mathbf{V} . For filter-project queries we adopt a different interpretation: although the *body* atom of a rewriting must reference a view in \mathbf{V} , the filter atoms can reference arbitrary base relations.

Definition 5.1.2. *A rewriting using \mathbf{V} is a query*

$$Q(\bar{t}) :- B \bowtie F$$

whose body atom references a view in \mathbf{V} and whose filter atoms reference base relations.

For reasons that will become clear later in this section, our goal is to place an upper bound on the information revealed by a query's body atom, without any concern for the information revealed by the query's filter atoms. Filter atoms remove rows from a query's output, and therefore can only *decrease* the amount of information revealed about the query's main body atom.

Consider the following query, which is a rewriting using $\{V_1\}$.

$$Q_{11}(u, n) :- V_1(u, n) \bowtie \text{Friend}(4, u)$$

Q_{11} reveals information about only a *subset* of the rows of V_1 : whereas V_1 reveals information about *arbitrary* users, Q_{11} only reveals information about friends of User 4. Apart from this distinction, the definition of rewritings for filter-project queries is analogous to the one used for ordinary conjunctive queries.

We next define query *expansions*. The expansion of a query Q is a new query Q_+ that returns the same answer as Q on any possible dataset, and whose body references a base relation rather than a view in \mathbf{V} . For instance, the expansion of Q_{11} above is

$$Q_{11+}(u, n) :- \text{User}(u, n, h) \times \text{Friend}(4, u)$$

Formally, the expansion of a query Q is obtained by taking the body atom $V(t_1, \dots, t_m)$ of Q and replacing each distinguished variable in V with the corresponding t_i . After performing the body atom of V becomes the body atom of Q , while the filter atoms of V become filter atoms of Q . Variables that are existential in V are renamed as needed to avoid capture.

We next argue for the correctness of the query expansion procedure defined above. If Q is a filter-project query whose body atom references a view V then $\text{BODYCOND}(Q)$ will be a conjunctive query combined semantics. The atom associated with V will be evaluated under bag semantics, while the remaining atoms will be evaluated under set semantics. Computing $\text{BODYCOND}(V)$, we obtain a view under combined semantics whose body atom is evaluated under bag semantics and whose filter atoms are evaluated under set semantics. Taking the expansion $\text{BODYCOND}(Q)_+$ of a query under combined semantics, as defined in Section 4.2, replaces the atom of $\text{BODYCOND}(Q)$ that is associated with V with the body of $\text{BODYCOND}(V)$, so that the body atom of V is evaluated under bag semantics and the remaining atoms (which correspond to the filter atoms of Q and of V) are evaluated under set semantics.

We now define a *rewriting of a query Q using a set of views \mathbf{V}* . As for ordinary

conjunctive queries, a rewriting Q^R of Q using \mathbf{V} has the same answer as Q on any possible dataset. We check the latter by determining whether $Q \equiv Q^R$. This criterion, which is based on the existence of *query homomorphisms*, is discussed in greater detail in Section 4.1.2.

Definition 5.1.3. *We say that Q^R is a rewriting of Q using \mathbf{V} if (i) Q^R is a rewriting using \mathbf{V} , and (ii) $\text{BODYCOND}(Q) \equiv \text{BODYCOND}(Q^R)$.*

In our running example, Q_{11} is a rewriting of V_2 using V_1 . This captures the intuition that V_1 reveals more information about its main body atom than V_2 , since V_1 reveals information about *arbitrary* users, whereas V_2 only reveals information about friends of the current user.

With this definition in place, we next shift our attention to the problem of determining whether a query Q has a rewriting Q' using \mathbf{V} . When searching for a rewriting, it is not immediately clear what filter atoms Q' should contain. The following Lemma allows us to circumvent this problem. It shows that we do not need to know up front what filter atoms should appear in Q' . Removing filter atoms from Q' will never cause condition (ii) to be violated, so we might as well assume when searching for queries satisfying condition (ii) that Q' does not contain *any* filter atoms.

Theorem 5.1.4. *Let Q be a filter-project query, and let \mathbf{V} be a set of filter-project views. The following are equivalent:*

- (i) *There exists a rewriting Q' using \mathbf{V} such that $\text{BODYCOND}(Q) \equiv \text{BODYCOND}(Q'_+)$.*
- (ii) *There exists a rewriting Q' using \mathbf{V} such that there are homomorphisms from $\text{BODY}(Q)$ to $\text{BODY}(Q'_+)$ and from $\text{BODYCOND}(Q'_+)$ to $\text{BODYCOND}(Q)$.*

Proof.

(i) implies (ii): Suppose (i) holds. Then there exists a rewriting Q' using \mathbf{V} such that $\text{BODYCOND}(Q) \equiv \text{BODYCOND}(Q'_+)$. Hence, we can find homomorphisms

$$\varphi : \text{BODYCOND}(Q) \rightarrow \text{BODYCOND}(Q'_+)$$

$$\text{and } \varphi' : \text{BODYCOND}(Q'_+) \rightarrow \text{BODYCOND}(Q)$$

Then φ maps $\text{BODY}(Q)$ to $\text{BODY}(Q'_+)$ and φ' maps $\text{BODYCOND}(Q'_+)$ to $\text{BODYCOND}(Q)$, so that condition (ii) is satisfied.

(ii) implies (i): Assume WLOG (by renaming variables as needed) that the heads of Q and Q' contain the same list of terms \bar{t} . Let Q'' be a query that contains the body of Q and the filter atoms of both Q and Q' . The distinguished variables of Q and Q' are the same, while the existential variables of Q are assumed to be disjoint from those of Q' .

We claim that $\text{BODYCOND}(Q) \equiv \text{BODYCOND}(Q''_+)$. Since Condition (ii) holds, there is a homomorphism φ from $\text{BODY}(Q)$ to $\text{BODY}(Q'_+)$. Since Q''_+ has the same head and body as Q'_+ , φ is also a homomorphism from $\text{BODY}(Q)$ to $\text{BODY}(Q''_+)$. By extending φ so that it maps every existential variable in the filter of Q to itself, we obtain a homomorphism from $\text{BODYCOND}(Q)$ to $\text{BODYCOND}(Q''_+)$.

In the other direction, Condition (ii) implies the existence of a homomorphism ψ' from $\text{BODYCOND}(Q'_+)$ to $\text{BODYCOND}(Q)$. By extending ψ' to map every existential variable in the filter of Q to itself, we obtain a homomorphism from $\text{BODYCOND}(Q''_+)$ to $\text{BODYCOND}(Q)$, and therefore $\text{BODYCOND}(Q) \equiv \text{BODYCOND}(Q''_+)$, as desired. \square

Theorem 5.1.4 provides a practical method for checking whether there is a rewriting of a query Q using a single view V . First, we search for a rewriting Q' of the single-atom conjunctive query $\text{BODY}(Q)$ using V ; in Section 4.3.2, we argued that this task can be performed quite efficiently. (See Proposition 4.2.7 for details.) If such a rewriting is found,

we check whether there is a homomorphism from $\text{BODYCOND}(Q'_+)$ to $\text{BODYCOND}(Q)$.

It is straightforward to generalize the algorithm to check for rewritings of a query Q using a set of views \mathbf{V} : we check, for each $V \in \mathbf{V}$, whether there is a rewriting of Q using V . A filter-project query can only have one body atom, so if Q has a rewriting using \mathbf{V} then it must have a rewriting using V for some view $V \in \mathbf{V}$.

We next consider the complexity of checking for rewritings.

Corollary 5.1.5. *Given a filter-project query Q and a set of filter-project views \mathbf{V} , the problem of determining whether there exists a rewriting of Q using \mathbf{V} is NP-complete.*

Proof. Given a rewriting Q^R of Q using \mathbf{V} we assume WLOG (using the previous Theorem) that the size of Q^R is polynomial in the sizes of Q and \mathbf{V} . Given homomorphisms

$$\begin{aligned}\varphi &: \text{BodyCond}(Q) \rightarrow \text{BodyCond}(Q'_+) \\ \varphi^R &: \text{BodyCond}(Q^R_+) \rightarrow \text{BodyCond}(Q)\end{aligned}$$

we can verify in polynomial time that (i) $\text{BODYCOND}(Q)$ and $\text{BODYCOND}(Q^R_+)$ are computed correctly and (ii) both of the homomorphisms defined above are valid. This means that the problem is in NP.

We now show that the problem is NP-hard. The proof is a reduction from the problem of checking the equivalence of conjunctive queries under set semantics, which is known to be NP-Hard. [19] Let

$$\begin{aligned}Q(\bar{t}) &:- A_1(\bar{t}_1) \wedge \dots \wedge A_n(\bar{t}_n) \\ \text{and } Q'(\bar{t}') &:- A'_1(\bar{t}'_1) \wedge \dots \wedge A_m(\bar{t}'_m)\end{aligned}$$

be conjunctive queries under set semantics, and assume WLOG that \bar{t} has the same arity

as \bar{t}' . Define filter-project queries

$$Q_*(\bar{t}) :- B(\bar{t}) \bowtie A_1(\bar{t}_1) \wedge \dots \wedge A_n(\bar{t}_n)$$

and $Q'_*(\bar{t}') :- B(\bar{t}') \bowtie A'_1(\bar{t}'_1) \wedge \dots \wedge A'_m(\bar{t}'_m)$

where B is a fresh relation with the same arity as \bar{t} and \bar{t}' . There is a trivial correspondence between multiset homomorphisms $\varphi : \text{BODYCOND}(Q) \rightarrow \text{BODYCOND}(Q')$ and set homomorphisms $\varphi_* : Q_* \rightarrow Q'_*$: the homomorphisms agree on all common terms. Similarly, there is a correspondence between multiset homomorphisms $\varphi' : \text{BODYCOND}(Q') \rightarrow \text{BODYCOND}(Q)$ and set homomorphisms $\varphi'_* : Q'_* \rightarrow Q_*$. Hence, $\text{BODYCOND}(Q)$ and $\text{BODYCOND}(Q')$ are homomorphic if and only if Q_* and Q'_* are homomorphic; NP-hardness follows. \square

5.1.2 Measuring Disclosure for Filter-Project Queries

We now show that the algorithms for *disclosure labeling* that we first introduced in Chapter 3 are applicable to filter-project queries. We begin by reviewing the definition of a *disclosure order* from Section 3.2.1. A *disclosure order* is a binary relation on sets of views. If \mathbf{V} and \mathbf{V}' are sets of filter-project queries then $\mathbf{V} \leq \mathbf{V}'$ means that the views in \mathbf{V}' reveal more information about their main body atoms than the views in \mathbf{V} .

Definition 5.1.6. A *disclosure order* is a binary relation \leq on subsets of \mathbf{U} such that

- (i) If $\mathbf{V}_1 \subseteq \mathbf{V}_2$ then $\mathbf{V}_1 \leq \mathbf{V}_2$.
- (ii) If $\mathbf{V}_1 \leq \mathbf{V}_2$ and $\mathbf{V}_2 \leq \mathbf{V}_3$ then $\mathbf{V}_1 \leq \mathbf{V}_3$.
- (iii) If $\mathbf{V}_1 \leq \mathbf{V}_3$ and $\mathbf{V}_2 \leq \mathbf{V}_3$ then $\mathbf{V}_1 \cup \mathbf{V}_2 \leq \mathbf{V}_3$.

If \mathbf{V}_1 and \mathbf{V}_2 are sets of filter-project views then we say that $\mathbf{V}_1 \leq \mathbf{V}_2$ if each element

of \mathbf{V}_1 has a homomorphic rewriting using \mathbf{V}_2 . Before proceeding any further, we must verify that this is actually a disclosure order.

Proposition 5.1.7. *The binary relation \leq defined above is a disclosure order.*

Proof. We verify each of the disclosure order properties in turn:

- If $\mathbf{V} \subseteq \mathbf{V}'$ then $\mathbf{V} \leq \mathbf{V}'$:

This follows immediately from the fact that each element of \mathbf{V} can be rewritten in terms of itself.

- If $\mathbf{V} \leq \mathbf{V}''$ and $\mathbf{V}' \leq \mathbf{V}''$ then $(\mathbf{V} \cup \mathbf{V}') \leq \mathbf{V}''$:

If $V \in \mathbf{V} \cup \mathbf{V}'$ then $V \in \mathbf{V}$ or $V \in \mathbf{V}'$; assume without loss of generality that $V \in \mathbf{V}$. By assumption, $\mathbf{V} \leq \mathbf{V}''$, and therefore V has an equivalent rewriting using \mathbf{V}'' .

- If $\mathbf{V} \leq \mathbf{V}'$ and $\mathbf{V}' \leq \mathbf{V}''$ then $\mathbf{V} \leq \mathbf{V}''$:

Each element of \mathbf{V} has a rewriting using \mathbf{V}' . We replace each body and condition in this rewriting with the rewriting of the corresponding element of \mathbf{V}' using \mathbf{V}'' . \square

In earlier Chapters we motivated disclosure orders using view determinacy: $\mathbf{V} \leq \mathbf{V}'$ meant that the answers to the views in \mathbf{V}' determined the answers to the views in \mathbf{V} on any possible dataset. That interpretation is *not* valid here. For instance, given the views

$$V_1(u, n) :- \text{User}(u, n, h)$$

$$V_2(u, n) :- \text{User}(u, n, h) \times \text{Friend}(4, u)$$

we previously noted that

$$Q_{11}(u, n) :- V_1(u, n) \times \text{Friend}(4, u)$$

is a rewriting of V_2 using $\{V_1\}$. However, the answer to V_2 is not always determined by the answer to V_1 , since V_2 depends on the contents of the `Friend` relation and V_1 does

not. Instead, \leq compares the information revealed about the main body atom of V_2 to the information revealed about the main body atom of V_1 . In the example above, V_1 reveals information about all the users in the dataset, whereas V_2 just reveals information about User 4's friends.

$\mathbf{V} \leq \mathbf{V}'$ does not generally imply that the views in \mathbf{V}' determine the answers to the views in \mathbf{V} . If the views in \mathbf{V} and \mathbf{V}' do not any contain filter atoms, however, then by Theorem 5.1.4, checking for rewritings of filter-project queries degenerates to checking for rewritings of conjunctive queries under bag semantics. In this case, the views in \mathbf{V}' *do* determine the answers to the views in \mathbf{V} . This observation agrees with our intuition that sets of views are ordered based on the amount of information they reveal about their main body atoms.

More generally, if $\mathbf{V} \leq \mathbf{V}'$ then each view $V \in \mathbf{V}$ has a rewriting using \mathbf{V}' . Any such rewriting contains the same columns of V and contains a superset of the rows. We next formalize this observation.

Proposition 5.1.8. *Suppose $\mathbf{V} \leq \mathbf{V}'$. For each $V \in \mathbf{V}$ there is a rewriting V' using \mathbf{V}' such that $Res(\text{BODYCOND}(V), \mathcal{D}) \subseteq Res(\text{BODYCOND}(V'), \mathcal{D})$ on all dataset \mathcal{D} .*

Proof. Fix $V \in \mathbf{V}$. By Theorem 5.1.4, there must exist a rewriting V' using \mathbf{V}' and homomorphisms $\varphi : \text{BODY}(V) \rightarrow \text{BODY}(V'_+)$ and $\varphi' : \text{BODYCOND}(V'_+) \rightarrow \text{BODYCOND}(V)$. Now φ maps the unique body atom of $\text{BODYCOND}(V)$ to the unique body atom of $\text{BODYCOND}(V')$. Hence, φ is bijective when restricted to terms in $\text{Body}(V)$; its inverse is φ' . Since multiset variables can only appear in the body atoms of $\text{BODYCOND}(V)$ and $\text{BODYCOND}(V'_+)$, φ must be bijective on multiset variables. By Theorem 4.1.5, we conclude that $Res(\text{BODYCOND}(V'_+), \mathcal{D}) \supseteq Res(\text{BODYCOND}(V), \mathcal{D})$ on all dataset \mathcal{D} . \square

We now show that filter-project queries are *decomposable*, so that $\{Q\} \leq \mathbf{V} \cup \mathbf{V}'$

implies $\{Q\} \leq \mathbf{V}$ or $\{Q\} \leq \mathbf{V}'$. This permits the use of efficient disclosure labeling algorithms that are discussed in detail in Section 3.3.2.

Corollary 5.1.9. *Suppose that Q is a filter-project query and that \mathbf{V} and \mathbf{V}' are sets of filter-project queries. If $\{Q\} \leq \mathbf{V} \cup \mathbf{V}'$ then $\{Q\} \leq \mathbf{V}$ or $\{Q\} \leq \mathbf{V}'$.*

Proof. Suppose that $\{Q\} \leq \mathbf{V} \cup \mathbf{V}'$. Then there exists a rewriting Q^S of Q using $\mathbf{V} \cup \mathbf{V}'$, and the unique body atom of Q^S must originate from either \mathbf{V} or \mathbf{V}' . If it originates from \mathbf{V} then $\{Q\} \leq \mathbf{V}$, as desired. Analogously, if it originates from \mathbf{V}' then $\{Q\} \leq \mathbf{V}'$. \square

5.2 SQL Query Language and Semantics

We now define a fragment of SQL that includes support for correlated subqueries, conjunction, disjunction, negation, and NULL values. In the discussion below, it will be useful to distinguish between *relations* (also referred to as *tables*) and *table instances*. Relations are query-independent and belong to the underlying dataset. *Table instances*, on the other hand, originate in the FROM clauses of SQL queries. Thus, a query containing a self-join has two table instances yet references just one relation. The query

```
SELECT U.uid, U.name FROM User U, Friend F
WHERE U.uid1 = 4 AND F.uid2 = U.uid
```

has two table instances: U and F.

5.2.1 Terms, Formulas, and Queries

We begin by defining a language that captures an interesting subset of the features of SQL. *Constants* are drawn from a finite set \mathbf{C} that contains at least one elements. We also

assume the existence of a distinguished constant NULL that does not appear in \mathbf{C} . A *value* v is an element of $\mathbf{C} \cup \{\text{NULL}\}$. A *term* t is either a value or a fully qualified table column name of the form T.Col, where T is a table instance and Col is a column within that table.

$$t ::= v \mid \text{T.Col}$$

A *formula* φ is either an equality comparison between two terms, the conjunction or disjunction of two formulas, the negation of a formula, or an EXISTS block that takes a correlated subquery Q as an argument.

$$\varphi ::= (t = t) \mid (\varphi \text{ AND } \varphi) \mid (\varphi \text{ OR } \varphi) \mid \text{NOT } (\varphi) \mid \text{EXISTS } (Q)$$

Fix an arbitrary constant $c \in \mathbf{C}$. We write TRUE as shorthand for the formula $(c = c)$, and we write FALSE as shorthand for NOT $(c = c)$. A query takes the form

```
SELECT  $A_1$  AS  $N_1$ , ...,  $A_n$  AS  $N_n$ 
FROM  $R_1$  AS  $T_1$ , ...,  $R_m$  AS  $T_m$ 
WHERE  $\varphi$ 
```

where R_1, \dots, R_m are base relations, T_1, \dots, T_m identify table instances, φ is a formula, and A_1, \dots, A_n are terms. We assume for the sake of simplicity that $m, n \geq 1$. The following is an example of a query in our language:

```
SELECT U.name AS myName, U.email AS myEmail
FROM User AS U
WHERE EXISTS (SELECT 1 AS one FROM Friend AS F
              WHERE U.uid = F.uid1 AND F.uid2 = 4)
```

In order to avoid issues related to variable capture when performing query evaluation, we additionally assume that every T_i is globally unique within a query. For example, the following query is not permitted in our fragment of SQL

```
SELECT U.name AS User U
WHERE EXISTS (SELECT U.uid AS uid FROM User U)
```

The reason is that there are two different table instances that are both named U.

$$\begin{aligned} \llbracket v \rrbracket_E^{\mathcal{D}} &= v \\ \llbracket T.Col \rrbracket_E^{\mathcal{D}} &= E(T.Col), \text{ where } T.Col \in \text{dom}(E) \end{aligned}$$

Evaluation semantics for terms.

$$\begin{aligned} \llbracket x = y \rrbracket_E^{\mathcal{D}} &= \begin{cases} \text{unknown}, & \text{if } \llbracket x \rrbracket_E^{\mathcal{D}} \text{ is null or } \llbracket y \rrbracket_E^{\mathcal{D}} \text{ is null} \\ \text{true}, & \text{if } \llbracket x \rrbracket_E^{\mathcal{D}} \text{ and } \llbracket y \rrbracket_E^{\mathcal{D}} \text{ are both non-null and equal} \\ \text{false}, & \text{if } \llbracket x \rrbracket_E^{\mathcal{D}} \text{ and } \llbracket y \rrbracket_E^{\mathcal{D}} \text{ are both non-null and not equal} \end{cases} \\ \llbracket \psi \text{ AND } \eta \rrbracket_E^{\mathcal{D}} &= \llbracket \psi \rrbracket_E^{\mathcal{D}} \text{ AND } \llbracket \eta \rrbracket_E^{\mathcal{D}} \\ \llbracket \psi \text{ OR } \eta \rrbracket_E^{\mathcal{D}} &= \llbracket \psi \rrbracket_E^{\mathcal{D}} \text{ OR } \llbracket \eta \rrbracket_E^{\mathcal{D}} \\ \llbracket \text{NOT } \psi \rrbracket_E^{\mathcal{D}} &= \text{NOT } \llbracket \psi \rrbracket_E^{\mathcal{D}} \\ \llbracket \text{EXISTS } Q \rrbracket_E^{\mathcal{D}} &= \begin{cases} \text{true}, & \text{if } \llbracket Q \rrbracket_E^{\mathcal{D}} \text{ is not empty} \\ \text{false}, & \text{otherwise} \end{cases} \end{aligned}$$

Evaluation semantics for formulas

Figure 5.2: Evaluation semantics for SQL terms and formulas.

5.2.2 Query Evaluation

Evaluation of terms, formulas, and queries, is parameterized on an *environment* E and a dataset \mathcal{D} . An *environment* E maps fully qualified table column names to values, while a dataset maps relation names to multisets of tuples.

Every term evaluates to a value, whereas every formula evaluates to one of the constants *true*, *false*, or *unknown*, which are interpreted under a three-valued Boolean logic. Evaluation semantics for terms and formulas are shown in Figure 5.2. The semantics for formulas rely on the definitions of AND, OR, and NOT in three-valued logic shown in Figure 5.3.

A *relational atom* has the form $A(s_1, \dots, s_n)$, where A is a predicate of arity n and each s_i is a (possibly NULL) constant. A dataset \mathcal{D} is a multiset containing atoms of the

	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

AND

	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

OR

T	F
F	T
U	U

NOT

Figure 5.3: Truth tables for three-valued Boolean logic.

uid	name	email
4	Zuck	zuck@fb.com
10	Marcel	marcel@fb.com
12347	Lucja	lucja@cornell.edu

User

uid1	uid2
4	10
10	4

Friend

Figure 5.4: Sample dataset based on Facebook Apps.

form $A(c_1, \dots, c_n)$ where A is a predicate of arity n and each c_i is a constant. We assume the existence of a *schema* that associates a unique name with each column $i = 1, 2, \dots, n$. For instance, consider the sample dataset shown in Figure 5.4. Formally, this dataset consists of a multiset of tuples

{ { User(4, Zuck, zuck@fb.com),
 User(10, Marcel, marcel@fb.com),
 User(12347, Lucja, lucja@cornell.edu),
 Friend(4, 10),
 Friend(10, 4) } }

together with the schema (uid, name, email) for the User relation and the schema (uid1, uid2) for the Friend relation.

Given a relation name R , we write $\mathcal{D}(R)$ to denote the multiset containing all the tuples in \mathcal{D} that are associated with the relation R . In the example above, we have

$$\mathcal{D}(\text{Friend}) = \{ (4, 10), (10, 4) \}$$

Given an environment E and a tuple τ we write $E[T \mapsto \tau]$ to denote the environment obtained by extending E with a fully qualified (column name, value) pair for each argument of τ . Let E be the environment

$$\{ \text{F.uid1} \mapsto 4, \text{F.uid2} \mapsto 10 \}$$

and let τ be the tuple

$$(4, \text{Zuck}, \text{zuck@fb.com})$$

from the User relation. Then $E[\text{U} \mapsto \tau]$ is the environment

$$\{ \text{U.uid} \mapsto 4, \text{U.name} \mapsto \text{Zuck}, \text{U.email} \mapsto \text{zuck@fb.com}, \\ \text{F.uid1} \mapsto 4, \text{F.uid2} \mapsto 10 \}$$

When we evaluate a database query, the result is a table. We refer to it as a *virtual table* in order to distinguish it from a *base table* such as User or Friend that appears in the database schema. Evaluating a query of the form

```
SELECT  $A_1$  AS  $N_1$ , ...,  $A_n$  AS  $N_n$ 
FROM  $R_1$  AS  $T_1$ , ...,  $R_m$  AS  $T_m$ 
WHERE  $\varphi$ 
```

yields a virtual table with schema (N_1, \dots, N_n) . The table's rows are computed as follows:

- (i) We evaluate the FROM clause by iterating all $\tau_1 \in \mathcal{D}(R_1), \dots, \tau_m \in \mathcal{D}(R_m)$. For each assignment of τ_1, \dots, τ_m we extend E with a new (key, value) pair for every argument of each τ_i to obtain a new environment E' .
- (ii) We evaluate the query's WHERE clause predicate under E' . If it evaluates to *false* or *undefined* then we do nothing. If it evaluates to *true* then we evaluate A_1, \dots, A_n to obtain values v_1, \dots, v_n , and then emit the tuple (v_1, \dots, v_n) .

Evaluation semantics for the process described above are formalized as follows:

$$\begin{aligned} \llbracket Q \rrbracket_E^{\mathcal{D}} = \{ & (\llbracket A_1 \rrbracket_{E'}^{\mathcal{D}}, \dots, \llbracket A_n \rrbracket_{E'}^{\mathcal{D}}) \\ & | \quad \tau_1 \in R_1, \dots, \tau_m \in R_m, \\ & E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m], \\ & \llbracket \varphi \rrbracket_{E'}^{\mathcal{D}} \text{ is satisfied } \} \end{aligned}$$

We now present an example to illustrate how query evaluation works. Let Q be the query

```
SELECT U.name AS myName, U.email AS myEmail
FROM User AS U
WHERE EXISTS (SELECT 1 AS one FROM Friend AS F
              WHERE U.uid = F.uid1 AND F.uid2 = 4)
```

Suppose we wish to evaluate Q against an empty environment and the dataset shown in Figure 5.4. We begin by iterating over tuples $\tau_U \in \mathcal{D}(\text{User})$. For each such τ_U we set $E' = \emptyset[U \mapsto \tau_U]$. For the first tuple in `User` we end up with the environment

$$E' = \{ \text{U.uid} \mapsto 4, \text{U.name} \mapsto \text{Zuck}, \text{U.email} \mapsto \text{zuck@fb.com} \}$$

We then evaluate the `WHERE` clause predicate under E' . To do so, we must recursively evaluate the subquery under E' . We iterate over atoms $\tau_F \in \mathcal{D}(\text{Friend})$ and create a new environment $E'' = E[F \mapsto \tau_F]$ for each one. For the first tuple in `Friend` we have

$$\begin{aligned} E'' = \{ \text{U.uid} \mapsto 4, \text{U.name} \mapsto \text{Zuck}, \text{U.email} \mapsto \text{zuck@fb.com}, \\ \text{F.uid1} \mapsto 4, \text{F.uid2} \mapsto 10 \} \end{aligned}$$

We next evaluate the subquery's `WHERE` clause predicate (`U.uid = F.uid1 AND F.uid2 = 4`) under E'' . In our running example, `U.uid` evaluates to 4 and `F.uid1` evaluates to 4, so (`U.uid = F.uid1`) evaluates to *true*. Similarly, `F.uid2` evaluates to 10, so (`F.uid2 = 4`) evaluates to *false*. Hence, the predicate is not satisfied, and no tuple is emitted for this assignment of τ_F . Similarly, no tuple is emitted when τ_F is set to

be the second tuple in the `Friend` relation. The subquery's answer is therefore empty when evaluated under the environment E' , so the `EXISTS` predicate evaluates to *false*. Consequently, the outer query does not emit anything when τ_U is the first tuple in `User`.

We next set τ_U to be the second tuple of the `User` relation and update E' accordingly. We then evaluate the subquery against this updated E' . This time, we find that setting $E'' = E'[F \mapsto \tau_U]$ causes the subquery's `WHERE` clause to be satisfied when τ_F is the second tuple of the `Friend` relation. The final answer of the subquery is a multiset containing a single tuple: $\{(1)\}$. The outer query's `EXISTS` clause evaluates to *true* because the subquery's answer is non-empty, so the outer query emits the tuple

(Marcel, marcel@fb.com)

Query evaluation proceeds in this manner. The `EXISTS` clause is not satisfied when τ_U is the third tuple in the `User` relation, so the outer query's final answer is the multiset

$\{(Marcel, marcel@fb.com)\}$

which contains a single tuple.

5.2.3 Free Variables and Safety

Query evaluation can fail if a query tries to access a table column that isn't in defined by the environment. For instance, attempting to evaluate the query

```
SELECT 1 AS one FROM Friend AS F
WHERE U.uid = F.uid1 AND F.uid2 = 4
```

against an empty environment will fail. More specifically, our attempt to evaluate the predicate $(U.uid = F.uid1)$ will fail because the environment under which it is evaluated does not associate a value with `U.uid`.

$$fv(v) = \emptyset$$

$$fv(T.Col) = T.Col$$

Free variables in terms.

$$fv(x = y) = fv(x) \cup fv(y)$$

$$fv(\psi \text{ AND } \eta) = fv(\psi) \cup fv(\eta)$$

$$fv(\psi \text{ OR } \eta) = fv(\psi) \cup fv(\eta)$$

$$fv(\text{NOT } \psi) = fv(\psi)$$

$$fv(\text{EXISTS } Q) = fv(Q)$$

Free variables in formulas.

Figure 5.5: Rules for computing free variables in terms and formulas.

In order to prevent such situations, we permit a query Q to be evaluated against an environment E only when the domain of E , denoted $dom(E)$, contains bindings for all the *free variables* in Q , denoted $fv(Q)$. Rules for computing free variables for terms and formulas are shown in Figure 5.5. For a query Q of the form

```

SELECT  $A_1$  AS  $N_1$ , ...,  $A_n$  AS  $N_n$ 
FROM  $R_1$  AS  $T_1$ , ...,  $R_m$  AS  $T_m$ 
WHERE  $\varphi$ 

```

we compute $fv(Q)$ as follows:

$$fv(Q) = (fv(\varphi) \cup fv(A_1) \cup \dots \cup fv(A_n)) - (T_1.* \cup \dots \cup T_m.*)$$

In other words, we first find all the free variables in φ and each A_i , and then remove all the columns of each T_i (denoted as $T_i.*$). For example, if Q is the example query defined above then $fv(Q) = \{U.uid\}$.

The evaluation of a query Q against an environment E is said to be *safe* if it never requires us to compute $E(T.Col)$ unless $T.Col$ is in the domain of E . We now present a criterion that is sufficient to ensure that query evaluation is safe.

Proposition 5.2.1. *Let E be an environment, and let X be a query, term, or formula. If $fV(X) \subseteq \text{dom}(E)$ then $\llbracket X \rrbracket_E^{\mathcal{D}}$ is safe for any dataset \mathcal{D} .*

Proof. The proof is by induction on terms, formulas, and queries. Consider a term t . If t is a value v then $\llbracket v \rrbracket_E^{\mathcal{D}}$ does not depend on E , so we're done. If t is a column reference of the form $T.\text{Col}$ then $fV(T.\text{Col}) = \{T.\text{Col}\}$, so $T.\text{Col} \in \text{dom}(E)$ by hypothesis.

Consider a formula of the form $(x = y)$. By construction, the domain of E contains $fV(x)$ and $fV(y)$, so $\llbracket x \rrbracket_E^{\mathcal{D}}$ and $\llbracket y \rrbracket_E^{\mathcal{D}}$ are safe by induction. These two values are sufficient to compute $\llbracket x = y \rrbracket_E^{\mathcal{D}}$, so we're done. Analogous arguments can be applied for the remaining kinds of formulas.

Finally, consider a query Q of the form outlined above. In this case, φ and each A_i is evaluated against the environment $E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m]$ for tuples $\tau_1 \in R_1, \dots, \tau_m \in R_m$. Hence, the domain of E' contains $\text{dom}(E)$ in addition to all the columns of each T_i . It therefore contains $fV(\varphi)$, as well as each $fV(A_i)$. Hence, $\llbracket \varphi \rrbracket_{E'}^{\mathcal{D}}$ and $\llbracket A_i \rrbracket_{E'}^{\mathcal{D}}$ are safe by induction, and this is sufficient to compute $\llbracket Q \rrbracket_E^{\mathcal{D}}$. \square

5.3 Query Generalization

Given a SQL query Q , our goal is to find a set of filter-project views V_1, V_2, \dots whose answers jointly determine the answer to Q on every possible dataset. This is done in two passes. In the first pass, we replace each relation referenced in Q with a filter-project view. The transformation is performed in such a way that the updated query Q' will return the same answer as the original query Q on any possible dataset. If Q is

```
SELECT U.name AS name, F.uid2 AS uid2
FROM User AS U, Friend AS F
```

```
WHERE U.uid = F.uid1 AND F.uid2 = 4
```

then one possible choice for Q' is

```
SELECT U.name AS name, F.uid2 AS uid2
FROM VU AS U, VF AS F
WHERE U.uid = F.uid1 AND F.uid2 = 4
```

where V_U and V_F are defined as follows:

$$V_U(U.uid, U.name, U.email) :- \text{User}(U.uid, U.name, U.email)$$
$$V_F(F.uid1, F.uid2) :- \text{Friend}(F.uid1, F.uid2)$$

The transformation given in the example above is not particularly interesting: V_U reveals the entirety of the `User` relation and V_F reveals the entirety of the `Friend` relation. However, it is possible to choose V_U and V_F more intelligently by projecting out columns that are never referenced within the body of Q . In the example above, we could instead define V_U as

$$V_U(U.uid, U.name) :- \text{User}(U.uid, U.name, U.email)$$

Notice that this new V_U reveals less information about the dataset than the old one: the old V_U revealed information about all the columns of the `User` table, whereas the new one only reveals information about the `uid` and `name` columns but not the `email` column of the `User` table. This corresponds to the standard query optimization heuristic of pushing project as far down into a query evaluation tree as possible, and will not be discussed further in this section.

After performing this transformation, it is easy to show that the answers to all the views referenced by Q' and its correlated subqueries uniquely determine the answer to

Q' (and therefore to Q as well) on any possible dataset. In the example above, the views

$$V_U(U.\text{uid}, U.\text{name}) :- \text{User}(U.\text{uid}, U.\text{name}, U.\text{email})$$
$$V_F(F.\text{uid1}, F.\text{uid2}) :- \text{Friend}(F.\text{uid1}, F.\text{uid2})$$

uniquely determine the answer to Q on any possible dataset. We refer to $\{V_U, V_F\}$ as *generalization* of Q , since the set represents a conservative upper bound on the information needed to answer Q .

Definition 5.3.1 (Query Generalization). *A generalization of a SQL query Q is a set of filter-project queries \mathbf{V} that determine the answer to Q on any possible dataset.*

Notice that there are some important facets of the original query Q that V_U and V_F do not capture. V_U reveals information about *all* the users in the dataset, whereas Q only reveals information about friends of User 4. Similarly, V_F reveals information about *arbitrary* friendships, whereas Q only reveals information about friendships of User 4. Such distinctions are essential to Facebook’s security model. However, they are not captured by the views V_U and V_F shown above.

The basic problem is that although V_U and V_F tell us which *columns* of the `User` and `Friend` tables are needed to answer Q , they tell us nothing about which *rows* of these tables are needed. Generating filter-project queries that have *column projections* is straightforward. However, the orthogonal problem of determining what *row selection predicates* or *filter atoms* a filter-project query should have is far more challenging. The remainder of this section is dedicated to the problem of finding “good” selection predicates and filters for queries.

5.3.1 Predicate Generalization

We start by considering the problem of determining which rows of a relation are needed to answer a given SQL query. For a running example, we will continue to use the SQL query Q defined as

```
SELECT U.name AS name, F.uid2 AS uid2
FROM User AS U, Friend AS F
WHERE U.uid = F.uid1 AND F.uid2 = 4
```

In the previous section we rewrote Q as

```
SELECT U.name AS name, F.uid2 AS uid2
FROM  $V_U$  AS U,  $V_F$  AS F
WHERE U.uid = F.uid1 AND F.uid2 = 4
```

where V_U and V_F had the bodies

$$\text{User}(U.\text{uid}, U.\text{name}, U.\text{email})$$

and

$$\text{Friend}(F.\text{uid1}, F.\text{uid2})$$

respectively. We can do better by adding new filter atoms to V_U and V_F to get

$$\begin{aligned} &\text{User}(U.\text{uid}, U.\text{name}, U.\text{email}) \times \text{Friend}(F.\text{uid1}, F.\text{uid2}) \\ &\quad \wedge (U.\text{uid} = F.\text{uid1}) \wedge (F.\text{uid2} = 4) \\ &\text{Friend}(F.\text{uid1}, F.\text{uid2}) \times \text{User}(U.\text{uid}, U.\text{name}, U.\text{email}) \\ &\quad \wedge (U.\text{uid} = F.\text{uid1}) \wedge (F.\text{uid2} = 4) \end{aligned}$$

Rather than revealing the entirety of the User and Friend relations, V_U only reveals a subset of the tuples in U. Similarly, V_F reveals information about only a subset of the tuples in F.

We now argue that this rewriting is correct, in the sense that Q' still returns the same answer as Q on any possible dataset. A tuple $\tau_U \in \mathcal{D}(\text{User})$ can only affect the answer to Q if there is a corresponding tuple $\tau_F \in \mathcal{D}(\text{Friend})$ such that $E[U \mapsto \tau_U, F \mapsto \tau_F]$ satisfies the SQL predicate

$$(U.\text{uid} = F.\text{uid1} \text{ AND } F.\text{uid2} = 4)$$

In this case, the environment $E[U \mapsto \tau_U, F \mapsto \tau_F]$ also satisfies the filter predicate

$$\text{Friend}(F.\text{uid1}, F.\text{uid2}) \wedge (U.\text{uid} = F.\text{uid1}) \wedge (F.\text{uid2} = 4)$$

of V_U , so the answer to V_U will contain a row for τ_U . Hence, V_U contains a row for every tuple in U whose presence or absence can affect the answer to Q . An analogous argument applies to F .

One subtlety in the argument above is that equality involving NULL values is handled differently for filter-project queries than it is for SQL queries. In SQL, $(\text{NULL} = \text{NULL})$ evaluates to *unknown*. Filter-project queries treat NULL just like any other constant, so that $(\text{NULL} = \text{NULL})$ evaluates to *true*. The translation we proposed works because equality in SQL implies equality in the filter-project query language: if $(\text{NULL} = \text{NULL})$ under SQL semantics then it is also true under filter-project semantics. The converse need not hold.

The same idea approach can be applied to arbitrary select-project-join query blocks with the help of the following Proposition. Given a table instance T_i we write \overline{T}_i to denote the list of fully qualified column names that appear in \overline{T}_i . For instance, if U is a table instance associated with the `User` relation, as in the example above, then \overline{U} is the triple $(U.\text{uid}, U.\text{name}, U.\text{email})$.

Proposition 5.3.2. *Consider a query Q of the form*

```

SELECT  $A_1$  AS  $N_1$ , ...,  $A_n$  AS  $N_n$ 
FROM  $R_1$  AS  $T_1$ , ...,  $R_m$  AS  $T_m$ 
WHERE  $\varphi$ 

```

We obtain a new query Q' by replacing each relation R_i with a conjunctive view V_{R_i} whose body is $R_i(\overline{T_i})$ and whose filter is

$$R_1(\overline{T_1}) \wedge \dots \wedge R_{i-1}(\overline{T_{i-1}}) \wedge R_{i+1}(\overline{T_{i+1}}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge \psi$$

Fix an environment E and a dataset \mathcal{D} such that, for all $E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m]$, if $\llbracket \varphi \rrbracket_E^{\mathcal{D}}$ is true then E' is satisfiably extendable w.r.t. ψ on \mathcal{D} . Then $\llbracket Q \rrbracket_E^{\mathcal{D}} = \llbracket Q' \rrbracket_E^{\mathcal{D}}$.

Proof. Consider an assignment of tuples $\tau_1 \in V_{R_1}, \dots, \tau_m \in V_{R_m}$ such that $\llbracket \varphi \rrbracket_{E'}^{\mathcal{D}}$ is true, where $E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m]$. By hypothesis, E' is a satisfying assignment of ψ , and E' satisfies $R_j(\overline{T_j})$ for each $j \neq i$ because $\tau_j \in R_j$ by construction. It follows that

$$R_1(\overline{T_1}) \wedge \dots \wedge R_{i-1}(\overline{T_{i-1}}) \wedge R_{i+1}(\overline{T_{i+1}}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge \psi$$

evaluates to *true* under E' . Since $E[T_i \mapsto \tau_i]$ can be extended to E' , it is satisfiably extendable w.r.t. ψ . This means that the multiplicity of τ_i is exactly the same in R_i and V_{R_i} . It follows that every tuple emitted by Q is also emitted by Q' . Since each V_{R_i} contains a subset of the tuples in the corresponding R_i , the converse also holds, and so Q and Q' emit exactly the same tuples. \square

Proposition 5.3.2 applies even for queries that contain self-joins. Let Q be the query

```
SELECT F2.uid2
FROM Friend F1, Friend F2
WHERE F1.uid1 = 4 AND F1.uid2 = F2.uid1
```

which outputs friends of friend of User 4. We can rewriting Q as

```
SELECT F2.uid2
FROM V_{F1} F1, V_{F2} F2
WHERE F1.uid1 = 4 AND F1.uid2 = F2.uid1
```

with the respective bodies and filters of V_{F_1} and V_{F_2} defined by

$$\begin{aligned} & \text{Friend}(F1.\text{uid1}, F1.\text{uid2}) \times \text{Friend}(F2.\text{uid1}, F2.\text{uid2}) \\ & \quad \wedge (F1.\text{uid1} = 4) \wedge (F1.\text{uid2} = F2.\text{uid1}) \\ & \text{Friend}(F2.\text{uid1}, F2.\text{uid2}) \times \text{Friend}(F1.\text{uid1}, F1.\text{uid2}), \\ & \quad \wedge (F1.\text{uid1} = 4) \wedge (F1.\text{uid2} = F2.\text{uid1}) \end{aligned}$$

If there are subqueries in the WHERE clause of Q then we must replace the relations in the FROM clauses of those subqueries with filter-project views.

5.3.2 Automated Query Rewriting

In the discussion above we proposed a criterion that allowed each table referenced by a SQL query to be replaced with a filter-project view that only revealed information about a subset of the rows and columns in that table. However, we did not say how such views could be computed. We now discuss a procedure for *automatically* generating a set of views that satisfies our criterion.

As discussed above, given the SQL query

```
SELECT A1 AS N1, ..., An AS Nn
FROM R1 AS T1, ..., Rm AS Tm
WHERE φ
```

we generate a filter-project view with body $R_i(\overline{T_i})$ and filter

$$R_1(\overline{T_1}) \wedge \dots \wedge R_{i-1}(\overline{T_{i-1}}) \wedge R_{i+1}(\overline{T_{i+1}}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge \psi$$

for each relation R_i . The predicate ψ is in principle arbitrary, subject to two restrictions. First, it must be a conjunction in the filter-project query language. More precisely, ψ must be a conjunction of predicates, each of which is either an atom $R'(\overline{T'})$ or an equality

constraint $(x = y)$ between bound variables and/or constants. And second, ψ must be satisfiably extendable whenever φ is true. We refer to ψ as a *generalization* of φ .

A generalization always exists: setting ψ to the constant *true* will always satisfy our constraints, no matter how φ is defined. This choice of ψ conservatively assumes that every row of every R_i can affect the answer to Q . We endeavor, however, to do something smarter when we can. We now define a recursive function $gen(\varphi)$ that computes a generalization for φ in a manner that satisfies our desiderata.

$$\begin{aligned} gen(x = y) &::= (x = y) \\ gen(\psi \text{ AND } \eta) &::= gen(\psi) \wedge gen(\eta) \\ gen(\psi \text{ OR } \eta) &::= true \\ gen(\text{NOT } \psi) &::= true \end{aligned}$$

The rule for an EXISTS clause is a bit more involved. Let S denote the query

```
SELECT A1 AS N1, ..., An AS Nn
FROM R1 AS T1, ..., Rm AS Tm
WHERE  $\varphi$ 
```

Then $gen(S)$ is

$$R_1(\overline{T_1}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge gen(\varphi)$$

We now verify that $gen(\varphi)$ always satisfies the criteria of Proposition 5.3.2.

Proposition 5.3.3. *Fix a formula φ , an environment E , and a dataset \mathcal{D} such that $fv(\varphi) \subseteq dom(E')$. For all $E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m]$, if $\llbracket \varphi \rrbracket_{E'}^{\mathcal{D}}$ is true then E' is satisfiably extendable w.r.t. $gen(\varphi)$ on \mathcal{D} .*

Proof. The proof is by structural induction. Suppose $\llbracket \varphi \rrbracket_{E'}^{\mathcal{D}}$ is true. If φ is of the form $(x = y)$ then $E(x) = E(y)$, so that E' satisfies the filter predicate $(x = y)$. If φ is of the form $(\psi \text{ OR } \eta)$ or $(\text{NOT } \psi)$ then $gen(\varphi) = true$, so E' trivially satisfies $gen(\varphi)$.

If φ is of the form $(\psi \text{ AND } \eta)$ then $\llbracket \psi \rrbracket_{E'}^{\mathcal{D}}$ and $\llbracket \eta \rrbracket_{E'}^{\mathcal{D}}$ must be true, so that E' is satisfiably extendable w.r.t. ψ and w.r.t. η by induction. This means that E' is extendable to environments E'_ψ (defined on all the variables in ψ) and E'_η (defined on all the variables in η) which satisfy $gen(\psi)$ and $gen(\eta)$ respectively. Since $fv(\varphi) \subseteq dom(E')$ and all table instance names are globally unique, E'_ψ and E'_η must agree with E' (and also with each other) on all common variables. Hence, we can merge them to obtain an environment E'' that satisfies $gen(\psi) \wedge gen(\eta)$. It follows that E' is satisfiably extendable w.r.t. $gen(\psi) \wedge gen(\eta)$ because E' is extendable to E'' .

Finally, suppose that the φ is of the form **EXISTS** S , where S is a subquery of the form described above. If $\llbracket \text{EXISTS } S \rrbracket_{E'}^{\mathcal{D}}$ is true then S must emit at least one tuple. If the relations in the **FROM** clause of S are R_1, \dots, R_m and the **WHERE** clause predicate is φ then there must be some $\tau_1 \in R_1, \dots, \tau_m \in R_m$ such that $\llbracket \varphi \rrbracket_{E'}^{\mathcal{D}}$ is true, where $E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m]$. By induction, $gen(\varphi)$ must be satisfiably extendable w.r.t. E' . By construction, $R_i(\overline{T_i})$ holds for each i , so E' is satisfiably extendable w.r.t.

$$R_1(\overline{T_1}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge gen(\varphi)$$

as desired. E is satisfiably extendable as well, since it can be extended to E' . \square

We next turn our attention to the problem of rewriting SQL queries. As before, our goal is to replace each relation R_i in the **FROM** clause of a query Q and each of its subqueries with a filter-project view V_{R_i} that emits only a subset of the tuples in R_i . The rewriting is performed by a recursive function *rew* that takes two arguments. The first argument is a query or formula to rewrite, while the second is a predicate θ that is known to be true whenever the output of the query/formula is able to affect the final output.

Rules for formulas are as follows:

$$\begin{aligned}
 \text{rew}(x = y, \theta) &::= (x = y) \\
 \text{rew}(\psi \text{ AND } \eta, \theta) &::= \text{rew}(\psi, \theta) \text{ AND } \text{rew}(\eta, \theta) \\
 \text{rew}(\psi \text{ OR } \eta, \theta) &::= \text{rew}(\psi, \theta) \text{ OR } \text{rew}(\eta, \theta) \\
 \text{rew}(\text{NOT } \psi, \theta) &::= \text{NOT } \text{rew}(\psi, \theta) \\
 \text{rew}(\text{EXISTS } S, \theta) &::= \text{EXISTS } \text{rew}(S, \theta)
 \end{aligned}$$

The rule for handling queries is more involved. Intuitively, the reason for this complexity is that we simultaneously push constraints from Q down into its correlated subqueries and pull constraints from the subqueries up into Q . If we have a query Q of the form

```

SELECT A1 AS N1, ..., An AS Nn
FROM R1 AS T1, ..., Rm AS Tm
WHERE φ1 AND ... AND φk

```

where no φ_i is itself a conjunction then $\text{rew}(Q, \theta)$ is

```

SELECT A1 AS N1, ..., An AS Nn
FROM VR1 AS T1, ..., VRm AS Tm
WHERE rew(φ1, θ1) AND ... AND rew(φk, θk)

```

where each θ_i is equal to the conjunction

$$\theta \wedge R_1(\overline{T_1}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge \text{gen}(\varphi_1) \wedge \dots \wedge \text{gen}(\varphi_{i-1}) \wedge \text{gen}(\varphi_{i+1}) \wedge \dots \wedge \text{gen}(\varphi_k)$$

If R_i is a base relation then the view V_{R_i} is a view with body $R_i(\overline{T_i})$ and filter

$$\theta \wedge R_1(\overline{T_1}) \wedge \dots \wedge R_{i-1}(\overline{T_{i-1}}) \wedge R_{i+1}(\overline{T_{i+1}}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge \text{gen}(\varphi)$$

where φ is equal to $(\varphi_1 \text{ AND } \dots \text{ AND } \varphi_k)$. For example, if Q is the query

```

SELECT U.name FROM User AS U
WHERE EXISTS (SELECT 1 AS one FROM Friend AS F
WHERE F.uid1 = U.uid AND F.uid2 = 4)

```

then $rew(Q, true)$ will yield a query of the form

```
SELECT U.name FROM VU AS U
WHERE EXISTS (SELECT 1 AS one FROM VF AS F
              WHERE F.uid1 = U.uid AND F.uid2 = 4)
```

that replaces the User table with V_U and the Friend table with V_F . Generalizing the subquery yields the conjunction

$$\text{Friend}(F.\text{uid1}, F.\text{uid2}) \wedge (F.\text{uid1} = U.\text{uid}) \wedge (F.\text{uid2} = 4)$$

and this becomes the filter for V_U . Within the scope of the inner subquery, θ is defined as

$$\text{User}(U.\text{uid}, U.\text{name}, U.\text{email})$$

and therefore the filter for V_F is

$$\text{User}(U.\text{uid}, U.\text{name}, U.\text{email}) \wedge (F.\text{uid1} = U.\text{uid}) \wedge (F.\text{uid2} = 4)$$

The first atom of the filter comes from θ , while the remaining two atoms are obtained by generalizing the subquery's WHERE clause.

In order to prove the correctness of the rewriting procedure outlined above, it would be useful to show that if $\llbracket \varphi \rrbracket_E^{\mathcal{D}}$ is *false* or *unknown* then $\llbracket rew(\varphi, \theta) \rrbracket_E^{\mathcal{D}}$ is also *false* or *unknown*, no matter how θ is chosen. However, this might not be true if φ contains an antijoin. Rather than assuming that $\llbracket \varphi \rrbracket_E^{\mathcal{D}}$ evaluates to *false* or *unknown*, we assume that E is not satisfiably extendable w.r.t. $gen(\varphi)$. Proposition 5.3.3 tells us that this is a strictly stronger hypothesis.

Lemma 5.3.4. *Fix an environment E , a SQL formula φ , and a filter predicate θ . If E is not satisfiably extendable w.r.t. $gen(\varphi)$ on \mathcal{D} then $\llbracket rew(\varphi, \theta) \rrbracket_E^{\mathcal{D}}$ is false or unknown.*

Proof. The proof is by structural induction on φ . If φ is of the form $(x = y)$ then $rew(\varphi, \theta)$ takes the form $(x = y)$, so the proof is immediate. If φ is of the form $(\psi \text{ OR } \eta)$ or the form $(\text{NOT } \psi)$ then $\llbracket gen(\varphi) \rrbracket_E^{\mathcal{D}}$ is always *true*, so the Lemma is trivially satisfied.

Suppose φ is of the form $(\psi \text{ AND } \eta)$. Then $gen(\varphi)$ is of the form $gen(\psi) \wedge gen(\eta)$. Let E' be an extension of E that is defined on all the variables in $gen(\psi)$ and $gen(\eta)$. Then E' is not satisfied on $gen(\psi) \wedge gen(\eta)$, so either $gen(\psi)$ is not satisfied or else $gen(\eta)$ is not satisfied; assume WLOG that the former is true. By induction, $\llbracket rew(\psi, \theta) \rrbracket_{E'}^D$ is *false* or *unknown*, so that $\llbracket rew(\varphi, \theta) \rrbracket_{E'}^D$, which is equal to $\llbracket rew(\psi, \theta) \text{ AND } rew(\eta, \theta) \rrbracket_{E'}^D$, must also be *false* or *unknown*. This argument is true for every extension E' of E , so the hypothesis follows.

Finally, suppose φ is of the form $(\text{EXISTS } S)$ for some subquery S whose WHERE clause ψ is of the form $(\psi_1 \text{ AND } \dots \text{ AND } \psi_k)$ and whose FROM clause is of the form

$$R_1 \text{ AS } T_1, \dots, R_m \text{ AS } T_m$$

Then $gen(\varphi)$ is equal to

$$R_1(\overline{T_1}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge gen(\psi)$$

Let $E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m]$ for some $\tau_1 \in R_1, \dots, \tau_m \in R_m$, and let E'' be an extension of E' that is defined on every variable in $gen(\psi)$. Then E'' does satisfy $gen(\varphi)$ by hypothesis, but E'' satisfies $R_i(\overline{T_i})$ for each $i = 1, 2, \dots, m$. So it must be the case that E'' does not satisfy $gen(\psi)$. But $gen(\psi)$ is the of the form

$$gen(\psi_1) \wedge \dots \wedge gen(\psi_k)$$

so there must be some j such that E'' does not satisfy $gen(\psi_j)$. By induction, $\llbracket rew(\psi_j, \theta_j) \rrbracket_{E''}^D$ must be *false* or *unknown*. Since this argument holds for all extensions E'' of E' , the WHERE clause of $rew(S, \theta)$ is never satisfied, and therefore $\llbracket rew(S, \theta) \rrbracket_E^D$ is empty, and $\llbracket rew(\varphi, \theta) \rrbracket_E^D$ is *false*. \square

With the preceding Lemma in place, we are now ready to prove that the correctness of the rewriting procedure defined above. As usual, the proof is by structural induction.

The rewriting procedure works whenever E is an environment that satisfies θ . If θ is the constant *true* then the rewriting procedure will work for *any* environment. However, our handling of subqueries requires us to prove the hypothesis for arbitrary θ .

Theorem 5.3.5. *Fix a predicate θ and an environment E such that E is satisfiably extendable w.r.t. θ , and let X be a formula or query such that $\text{fv}(X) \subseteq \text{dom}(E)$. If E is defined on all the variables that are common to $\text{gen}(X)$ and θ then $\llbracket X \rrbracket_E^{\mathcal{D}} = \llbracket \text{rew}(X, \theta) \rrbracket_E^{\mathcal{D}}$.*

Proof. For formulas, the argument is straightforward. Given a formula φ of the form $(x = y)$ we have $\text{rew}(\varphi) = \varphi$, so the hypothesis holds trivially. Given a formula φ of the form $(\psi \text{ AND } \eta)$, we know by induction that $\llbracket \psi \rrbracket_E^{\mathcal{D}} = \llbracket \text{rew}(\psi, \theta) \rrbracket_E^{\mathcal{D}}$ and $\llbracket \eta \rrbracket_E^{\mathcal{D}} = \llbracket \text{rew}(\eta, \theta) \rrbracket_E^{\mathcal{D}}$, and therefore

$$\begin{aligned}
& \llbracket \text{rew}(\psi \text{ AND } \eta, \theta) \rrbracket_E^{\mathcal{D}} \\
&= \llbracket \text{rew}(\psi, \theta) \text{ AND } \text{rew}(\eta, \theta) \rrbracket_E^{\mathcal{D}} \\
&= \llbracket \text{rew}(\psi, \theta) \rrbracket_E^{\mathcal{D}} \text{ AND } \llbracket \text{rew}(\eta, \theta) \rrbracket_E^{\mathcal{D}} \\
&= \llbracket \psi \rrbracket_E^{\mathcal{D}} \text{ AND } \llbracket \eta \rrbracket_E^{\mathcal{D}} \\
&= \llbracket \psi \text{ AND } \eta \rrbracket_E^{\mathcal{D}}
\end{aligned}$$

The proof for OR and NOT statements is analogous. For a predicate φ of the form $(\text{EXISTS } S)$ we know by induction that $\llbracket \text{rew}(S, \theta) \rrbracket_E^{\mathcal{D}} = \llbracket S \rrbracket_E^{\mathcal{D}}$, so that the former is empty if and only if the latter is empty. We conclude that

$$\llbracket \text{rew}(\text{EXISTS } S, \theta) \rrbracket_E^{\mathcal{D}} = \llbracket \text{EXISTS } \text{rew}(S, \theta) \rrbracket_E^{\mathcal{D}} = \llbracket \text{EXISTS } S \rrbracket_E^{\mathcal{D}}$$

The argument for queries is more involved. Consider a query Q of the form described above, and let φ be equal to $(\varphi_1 \text{ AND } \dots \text{ AND } \varphi_k)$. Select tuples $\tau_1 \in R_1, \dots, \tau_m \in R_m$, and set $E' = E[T_1 \mapsto \tau_1, \dots, T_m \mapsto \tau_m]$. By Proposition 5.3.3, we know that E' is satisfiably extendable w.r.t. $\text{gen}(\varphi)$ whenever $\llbracket \varphi \rrbracket_{E'}^{\mathcal{D}}$ is true. Furthermore, E' is satisfiably

extendable w.r.t. θ by assumption. Since E' is defined on all variables common to $gen(\varphi)$ and θ , we conclude that if $\llbracket \varphi \rrbracket_{E'}^{\mathcal{D}}$ then E is satisfiably extendable w.r.t. $gen(\varphi) \wedge \theta$. By Proposition 5.3.2, it follows that replacing each R_i with the corresponding R_{V_i} does not change the query's answer.

It remains for us to show that the original WHERE clause and the rewritten WHERE clause are equivalent. More concretely, we claim that $\llbracket \varphi_1 \text{ AND } \dots \text{ AND } \varphi_k \rrbracket_{E'}^{\mathcal{D}}$ is true if and only if $\llbracket rew(\varphi_1, \theta_1) \text{ AND } \dots \text{ AND } rew(\varphi_k, \theta_k) \rrbracket_{E'}^{\mathcal{D}}$ is true. First suppose that E' is satisfiably extendable w.r.t. $gen(\varphi_i)$ for each $i = 1, 2, \dots, m$. In this case, E' is satisfiably extendable for each θ_i , and therefore $\llbracket \varphi_i \rrbracket_{E'}^{\mathcal{D}} = \llbracket rew(\varphi_i, \theta_i) \rrbracket_{E'}^{\mathcal{D}}$ for each i by induction. It follows that

$$\llbracket rew(\varphi_1, \theta_1) \text{ AND } \dots \text{ AND } rew(\varphi_k, \theta_k) \rrbracket_{E'}^{\mathcal{D}}$$

is true; this is the WHERE clause of the rewritten query.

Now suppose instead that E' is not satisfiably extendable w.r.t. $gen(\varphi_i)$ for some i . By the preceding Lemma, we conclude that $\llbracket rew(\varphi_i, \theta_i) \rrbracket_{E'}^{\mathcal{D}}$ is false or unknown as well. This means that the rewritten WHERE clause is false or unknown, and so performing the rewriting does not change the final answer. \square

5.3.3 Query Generalization and Disclosure Orders

Section 5.1.2 defines a disclosure order that allows us to compare the amounts of information revealed about the body atoms of two filter-project queries. There is a strong connection between this notion of information disclosure and the transformation from SQL to filter-project queries discussed in Section 5.3.1. In this section we explain and formalize the connection. Suppose that, as in the preceding example, Q is the query

```
SELECT U.name AS name, F.uid2 AS uid2
```

```

FROM User AS U, Friend AS F
WHERE U.uid = F.uid1 AND F.uid2 = 4

```

We showed earlier in this section that Q could be rewritten as

```

SELECT U.name AS name, F.uid2 AS uid2
FROM VU AS U, VF AS F
WHERE U.uid = F.uid1 AND F.uid2 = 4

```

where V_U and V_F are defined by

$$V_U(u, n) :- \text{User}(u, n, e) \times \text{Friend}(u_1, u_2) \wedge (u = u_1) \wedge (u_2 = 4)$$

$$V_F(u_1, u_2) :- \text{Friend}(u_1, u_2) \times \text{User}(u, n, e) \wedge (u = u_1) \wedge (u_2 = 4)$$

respectively.¹ The rewriting depends only on V_F and V_U , so the answers to V_F and V_U must uniquely determine the answer to Q on any possible dataset. However, we can show a much stronger result: if \mathbf{V}' is *any* set of filter-project queries such that $\{V_F, V_U\} \leq \mathbf{V}'$ then \mathbf{V}' also determines the answer to Q . For instance, if we define

$$V'_U(u, n, e) :- \text{User}(u, n, e)$$

$$V'_F(u_1, u_2) :- \text{Friend}(u_1, u_2)$$

then $\{V_U, V_F\} \leq \{V'_U, V'_F\}$, and therefore V'_U and V'_F uniquely determine the answer to Q on any possible dataset. This is true even though $\mathbf{V} \leq \mathbf{V}'$ does *not* generally imply that the answers to the views in \mathbf{V}' determine the answers to the views in \mathbf{V} . For example, although $\{V_U\} \leq \{V'_U\}$, it is not the case that V'_U determines the answer to V_U . In particular, V'_U does not depend on the `Friend` relation, whereas V_U does. However, $\{V'_U, V'_F\}$ still determines Q because V'_U tells us everything we need to know about the `User` relation to answer Q and V'_F tells us everything we need to know about the `Friend` relation. This intuition is formalized by the following result.

¹Rather than using fully qualified table column names as we did above, we use much shorter variable names to enhance readability.

Proposition 5.3.6. *Let Q be a SQL query, and let $\mathbf{V} = \{V_{R_1}, \dots, V_{R_m}\}$ be the views referenced in the FROM clauses of Q and its subqueries. If \mathbf{V}' is a set of views such that $\mathbf{V} \leq \mathbf{V}'$ then the views in \mathbf{V}' uniquely determine the answer to Q on any possible dataset.*

Proof. For each $V_{R_i} \in \mathbf{V}$ we have $\{V_{R_i}\} \leq \mathbf{V}'$, so there exists a rewriting V'_{R_i} of V_{R_i} using \mathbf{V}' . The transformation described in Section 5.3.1 replaces each reference to a relation R_i in Q with a view V_{R_i} that reveals a subset of the rows and columns of R_i . Since $\{V_{R_i}\} \leq \mathbf{V}'$, the view V_{R_i} must have a rewriting V'_{R_i} using \mathbf{V}' . By Proposition 5.1.8, we must have

$$Res(\text{BODYCOND}(V_{R_i}), \mathcal{D}) \subseteq Res(\text{BODYCOND}(V'_{R_i}), \mathcal{D})$$

for all dataset \mathcal{D} . This means that V'_{R_i} contains the same columns as V_{R_i} and contains a superset of the rows. So we can replace each V_{R_i} in the rewritten version of Q with V'_{R_i} without affecting the correctness of Proposition 5.3.2. \square

5.4 Condition Graphs

In the previous section we showed how to generate a collection of filter-project queries that acted as an upper bound on the information needed to compute a SQL query's answer. We now present a different algorithm that solves the same problem by constructing and then traversing a directed graph, which we refer to as a *condition graph*. For queries in the restricted fragment of SQL discussed above, the two algorithms are equivalent, so the condition graph algorithm is provably correct. The condition graph algorithm works with many features that do not appear in the fragment of SQL defined in Section 5.2; for queries containing such features we offer informal arguments of correctness. Our running example will reference the queries in Figure 5.6.

- (a) `SELECT U1.name FROM User U1, Friend F1
WHERE F1.uid1 = 1 AND F1.uid2 = U1.uid`
- (b) `SELECT U1.name
FROM User U1 LEFT OUTER JOIN Friend F1
ON (F1.uid1 = 1 AND F1.uid2 = U1.uid)`
- (c) `SELECT U1.name
FROM User U1 FULL OUTER JOIN Friend F1
ON (F1.uid1 = 1 AND F1.uid2 = U1.uid)`
- (d) `SELECT U1.name
FROM User U1, Friend F1, Friend F2
WHERE F1.uid1 = 1 AND F1.uid2 = F2.uid1
AND F2.uid2 = U1.uid`
- (e) `SELECT U1.name FROM User U1 WHERE U1.uid IN
(SELECT F1.uid2 FROM Friend F1
WHERE F1.uid1 = 1)`

Figure 5.6: SQL queries handled by our condition graph algorithm.

5.4.1 Constructing Condition Graphs

The condition graph for a query Q contains one node (labeled σ) for each SELECT statement in the query, one node (labeled \bowtie) for each two-way join in the query's logical evaluation plan, and one node for each table instance defined by the query. Figures 5.7a through 5.7e show the condition graphs for the SQL queries from Figures 5.6a through 5.6e respectively.

In many cases, condition graphs bear a deceptive resemblance to expression trees in the Relational Algebra. However, there are a number of important differences between the two. Edges in condition graphs can be unidirectional or bidirectional, and every condition graph must contain a SELECT node at its root. Furthermore, the same nodes appear in the graph regardless of whether we perform an inner join or an outer join. However, the directions of edges between those nodes can change depending on the type of join we are dealing with (Figures 5.7a, 5.7b, and 5.7c). And finally, SELECT nodes can have multiple children (Figure 5.7e). One child represents the FROM clause of

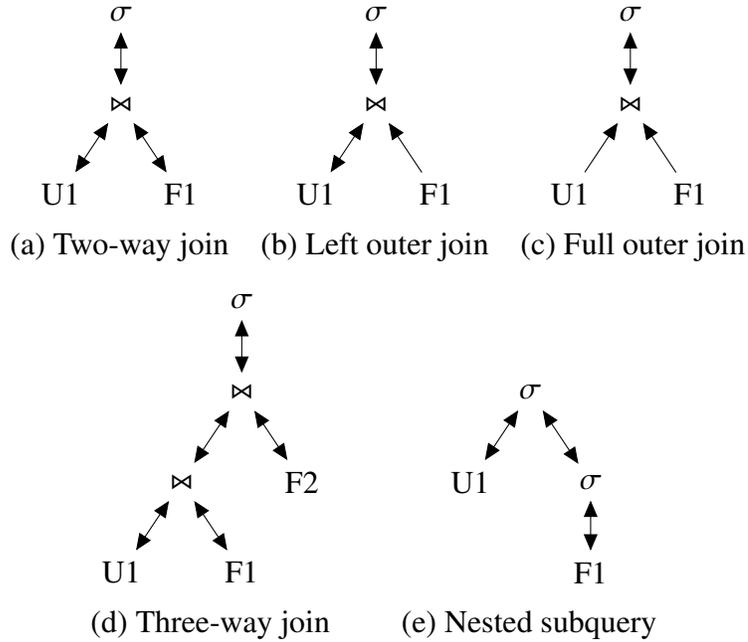


Figure 5.7: Condition graphs for queries from Fig. 5.6

the corresponding `SELECT` statement, while the remaining children represent correlated subqueries that appear in the statement's `WHERE` and `HAVING` clauses. Temporary tables (not shown here) are handled in a similar manner to nested subqueries.

Once we have constructed the condition graph we can determine which semijoin filter atoms should appear in each of the output queries. Our criterion is based on reachability in the condition graph: a directed path from table instance R to table instance S indicates that a tuple in R can only affect the query's output if it can be joined with a tuple from S . In our running example, Q_{U1} should contain a filter atom for $F1$ precisely when the node associated with $F1$ is reachable from the node associated with $U1$ in the condition graph. For the query in Figure 5.6a we obtain

$$Q_{U1}(u, n) \text{ :- } U(u, n, h) \bowtie F(u_1, u_2)$$

$$Q_{F1}(u_1, u_2) \text{ :- } F(u_1, u_2) \bowtie U(u, n, h)$$

The situation will be similar for the left outer join query in Figure 5.7b except that Q_{U1}

will not contain any filter atoms. For the full outer join query in Figure 5.7c neither Q_{U1} nor Q_{F1} will contain any filter atoms. An analogous criterion can be used for other types of queries, including those in Figures 5.7d and 5.7e.

The next step is to find equality constraints that should be added to Q_{U1} and Q_{F1} . We begin by finding all SELECT nodes that are reachable from the condition graph node associated with U1. We then check the WHERE and HAVING clauses of the corresponding SELECT statements for equality constraints that must hold for all tuples in the statements' outputs. In the query from Figure 5.6a, we obtain two constraints: $(F1.uid1 = 1)$ and $(F1.uid2 = U1.uid)$. When we integrate both constraints into Q_{U1} we obtain the filter-project query

$$Q_{U1}(u, n) :- U(u, n, h) \bowtie F(u_1, u_2) \wedge (u_1 = 1) \wedge (u_2 = u)$$

which can be simplified to

$$Q_{U1}(u, n) :- U(u, n, h) \bowtie F(1, u)$$

An analogous process can be performed with Q_{F1} to obtain

$$Q_{F1}(1, u_2) :- F(1, u_2) \bowtie U(u_2, n, h)$$

The filter-project queries Q_{U1} and Q_{F1} comprise the output of our extraction algorithm.

5.4.2 Correctness for Restricted SQL Queries

We now show that the condition graph algorithm discussed above is equivalent to the rewriting algorithm from Section 5.3.2 for queries expressed in the fragment of SQL defined in Section 5.2. Correctness of the condition graph algorithm then follows immediately from the correctness of the rewriting algorithm.

Recall that if Q is a query of the form

```
SELECT A1 AS N1, ..., An AS Nn
FROM R1 AS T1, ..., Rm AS Tm
WHERE φ1 AND ... AND φk
```

then $rew(Q, \theta)$ is

```
SELECT A1 AS N1, ..., An AS Nn
FROM VR1 AS T1, ..., VRm AS Tm
WHERE rew(φ1, θ1) AND ... AND rew(φk, θk)
```

where each θ_i is equal to the conjunction

$$\theta \wedge R_1(\overline{T_1}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge gen(\varphi_1) \wedge \dots \wedge gen(\varphi_{i-1}) \wedge gen(\varphi_{i+1}) \wedge \dots \wedge gen(\varphi_k)$$

and if R_i is a base relation then the view V_{R_i} is a view with body $R_i(\overline{T_i})$ and filter

$$\theta \wedge R_1(\overline{T_1}) \wedge \dots \wedge R_{i-1}(\overline{T_{i-1}}) \wedge R_{i+1}(\overline{T_{i+1}}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge gen(\varphi)$$

where φ is equal to $(\varphi_1 \text{ AND } \dots \text{ AND } \varphi_k)$.

The crux of our argument is that if R_i is a relation that appears in the FROM clause of Q then traversing the subtree rooted at φ_j is equivalent to appending $gen(\varphi_j)$ to the filter of V_{R_i} . Traversing the node for R_k when $k \neq i$ is equivalent to appending $R_k(\overline{T_k})$ to the filter of V_{R_i} . And traversing the remainder of the condition graph is equivalent to appending θ to the filter of V_{R_i} . So performing a traversal of the condition graph starting at R_i will yield the filter for V_{R_i} shown above.

Our proof relies on the following Lemma, which shows an equivalence between traversing a condition graph and generalizing SQL formulas. We write χ to denote the list of filter atoms in V_{R_i} .

Lemma 5.4.1. *Traversing a SELECT block S and all of its children is equivalent to appending $gen(EXISTS S)$ to χ .*

Proof. Suppose that the statement $(R_j \text{ AS } T_j)$ appears in the FROM clause of S . Then traversing the corresponding node in the condition graph will cause us to append the atom $R_j(\overline{T_j})$ to χ .

Now suppose that φ_j is a formula in the WHERE clause of S . If φ_j is of the form $(x = y)$ then the predicate $(x = y)$ will be appended to χ when we traverse the node associated with S ; this is equivalent to appending $gen(\varphi_j)$ to χ . By assumption, φ_j is *not* of the form $(\psi_1 \text{ AND } \psi_2)$, so we can ignore that case. If φ_j is of the form $(\psi_1 \text{ OR } \psi_2)$ or the form $(\text{NOT } \psi_1)$ then it will have no impact on χ ; furthermore, $gen(\varphi_j)$ is *true* in these cases. Finally, suppose φ_j is of the form EXISTS S' . By induction, traversing S' and all its children is equivalent to appending $gen(\text{EXISTS } S')$ to χ .

Combining these observations we see that traversing S and all of its children is equivalent to appending $gen(\text{EXISTS } S)$ to χ . □

We are now ready to prove equivalence of the condition graph traversal algorithm and the rewriting procedure defined above.

Proposition 5.4.2. *Fix a query Q , and suppose that the FROM clause of Q or one of its subqueries contains the statement $(R_i \text{ AS } T_i)$. The filter χ generated by traversing the query's condition graph from the node associated with R_i is the same as the filter created for V_{R_i} by $rew(Q, \text{true})$ up to reordering of conjuncts.*

Proof. The original condition graph traversal algorithm starts at the node associated with $(R_i \text{ AS } T_i)$ and visits every vertex that is reachable from that node (except the starting node itself). However, we consider a variant of the algorithm that visits exactly the same nodes in a different order. Our algorithm starts at the root SELECT block and descends recursively into its children. Our claim is that this traversal yields the correct value of χ . The actual traversal is the same except that nodes will be visited in a different order; this

can affect the order of the conjuncts in χ but will not affect the truth value of χ because conjunction is a commutative operation.

Starting from the root node S our algorithm first visits nodes corresponding to FROM clause statements in S . It then visits all the subtrees associated with the WHERE clause of S except for the one that contains $(R_i \text{ AS } T_i)$. This is equivalent to appending

$$R_1(\overline{T_1}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge \text{gen}(\varphi_1) \wedge \dots \wedge \text{gen}(\varphi_{i-1}) \wedge \text{gen}(\varphi_{i+1}) \wedge \dots \wedge \text{gen}(\varphi_k)$$

to χ . Finally, it descends into the last WHERE clause node; at that point, θ is equal to χ . The process is repeated as necessary: we may need to descend into multiple layers of correlated subqueries before we reach the SELECT block whose FROM clause contains $(R_i \text{ AS } T_i)$. At each step, θ and χ are the same up to reordering of conjuncts.

We will eventually reach the SELECT block S' that has $(R_i \text{ AS } S_i)$ in its FROM clause. We traverse every FROM clause node except for the one associated with $(R_i \text{ AS } S_i)$; this has the effect of appending

$$R_1(\overline{T_1}) \wedge \dots \wedge R_{i-1}(\overline{T_{i-1}}) \wedge R_{i+1}(\overline{T_{i+1}}) \wedge \dots \wedge R_m(\overline{T_m})$$

to χ . We then traverse S and all of its WHERE clause children; this has the effect of appending $\text{gen}(\varphi)$ to χ . At the end of the process, the nodes in χ are exactly

$$\theta \wedge R_1(\overline{T_1}) \wedge \dots \wedge R_{i-1}(\overline{T_{i-1}}) \wedge R_{i+1}(\overline{T_{i+1}}) \wedge \dots \wedge R_m(\overline{T_m}) \wedge \text{gen}(\varphi)$$

as desired. □

5.4.3 Correctness for Extended SQL

We now argue informally that condition graphs can be applied to a wide variety of SQL features beyond the ones in our core query language.

- Ambiguously named columns or unqualified column names can be disambiguated as a preprocessing step. `SELECT *` can be expanded to list the names of the referenced columns. And table instances can be assigned globally unique names. For example,

```
SELECT * FROM User WHERE uid = 4
```

can be rewritten as

```
SELECT U.uid AS uid, U.name AS name, U.hobby AS hobby
FROM User AS U
WHERE U.uid = 4
```

- When determining which filter atoms a view should have we can conservatively assume that a complex selection predicate always evaluates to *true*; making this assumption can only increase the number of tuples in the input relations that can affect the query's answer. For example, we can replace the query

```
SELECT * FROM User WHERE U.name = 'Alice' AND U.uid < 5
```

with the query

```
SELECT * FROM User WHERE U.name = 'Alice'
```

Any row in `U` that can affect the first query's answer can also affect the second query's answer. This argument, which can be applied to a wide variety of selection predicates, is an application of Proposition 5.3.2 above, and it naturally extends the way our query generalization algorithm handles negation and disjunction.

- Although we only consider select-project-join query blocks in our core language, most clauses that might appear in a `SELECT` statement, including `GROUP BY`, `HAVING`, aggregation, `ORDER BY`, and `LIMIT`, are logically evaluated *after* the query's `WHERE` clause. For instance, evaluating the query

```
SELECT COUNT(*) FROM User GROUP BY name
```

is logically equivalent to evaluating the much simpler query

```
SELECT name FROM User
```

and then counting the number of tuples associated with each *name*.

- Dealing with set operations such as $Q_1 \text{ UNION ALL } Q_2$ is conceptually straightforward: we compute a set of filter-project views that determines the answer to Q_1 and another set of views that determines the answer to Q_2 ; the union of these two sets must determine the answer to $Q_1 \text{ UNION ALL } Q_2$.
- Outer joins can be rewritten using semijoins and unions. For example, the query

```
SELECT U.uid AS uid, F.uid2 AS uid2
FROM User AS U LEFT OUTER JOIN Friend AS F
ON (U.uid = F.uid1)
```

can be rewritten as

```
(SELECT U1.uid AS uid, F1.uid2 AS uid2
FROM User AS U1, Friend AS F1
WHERE U.uid = F.uid1)
UNION ALL
(SELECT U2.uid AS uid, NULL AS uid2
FROM User AS U2
WHERE NOT EXISTS
(SELECT * FROM Friend AS F2 WHERE U2.uid = F2.uid1))
```

Notice that the filter-project queries generated for F1 and F2 will both contain filter atoms associated with the User relation; in fact, they will contain exactly the same filter atoms up to isomorphism. However, the filter-project view generated for U1 will contain a filter atom for the Friend relation, while the one associated with U2 will not. The filter-project query generated for U2 will contain strictly more tuples than the one generated for U1. Consequently, the filter-project queries generated for the query

```
SELECT U2.uid, NULL AS uid2
FROM User AS U2
WHERE NOT EXISTS
(SELECT * FROM Friend AS F2 WHERE U2.uid = F2.uid1)
```

will contain enough information to answer the original outer join on any possible dataset. This observation justifies the graph topology in Figure 5.7(b).

5.5 Summary

The results in this Chapter provide us with an end-to-end pipeline for reasoning about information disclosure in complex SQL queries. We first generalize the SQL queries to filter-project queries using the techniques discussed in Sections 5.3 and 5.4. If desired, we can subsequently use the analysis from Section 5.1 to efficiently label the resulting filter-project queries as a post-processing step.

CHAPTER 6

EXPLAINABLE SECURITY

In Chapter 3 we introduced *disclosure labeling*, which allowed us to reason about the information needed to answer a database query. The results of Chapter 3 were extended to richer query languages in Chapters 4 and 5. In this Chapter we discuss two applications of disclosure labeling. Our main application is *explainable security*, a new security model in which every security policy decision is accompanied by a formal justification, which we call an *explanation*. However, we also show at the end of this Chapter how the same results can be applied to the problem of *query pricing* proposed by Koutris et al. [35].

Our security guarantees are the same as in Chapter 3. A principal is granted access to a set of views \mathbf{P} , which we refer to as a *security policy*. He is then permitted to learn the answer to any query Q whose answer is uniquely determined by the views in \mathbf{P} on any possible dataset, so that $\{Q\} \leq \mathbf{P}$. In Section 6.1 we show how to construct a Boolean formula that precisely characterizes the policies which satisfy this condition. In Section 6.2 we show how these *policy formulas* can be used to generate explanations for policy decisions, and in Section 6.3 we discuss applications of policy formulas to query pricing.

6.1 Policy Formulas and Explanations

We start with a set of *security views* \mathbf{V} defined by a human administrator. Each view in \mathbf{V} roughly corresponds to a single *permission* that a principal may or may not be granted. A policy formula is a Boolean formula in which every security view is treated as a variable. *Policy formulas over \mathbf{V}* are formally defined using the BNF grammar

$$\tau ::= 0 \mid 1 \mid V \mid (\tau \vee \tau) \mid (\tau \wedge \tau)$$

$$\begin{aligned}
V_1(u, n, h) &:- \text{User}(u, n, h) \\
V_2(u, n) &:- \text{User}(u, n, h) \\
V_3(1, n, h) &:- \text{User}(1, n, h) \\
V_4(h) &:- \text{User}(u, n, h) \\
V_5(u_1, u_2) &:- \text{Friend}(u_1, u_2) \\
V_6(1, u_2) &:- \text{Friend}(1, u_2)
\end{aligned}$$

Figure 6.1: A set of security views

where each V is a security view in \mathbf{V} . The constant 0 corresponds to a policy in which a query's execution is never permitted, while the constant 1 corresponds to a policy in which a query's execution is always permitted.

Consider the set of security views in Figure 6.1. The policy formula V_1 indicates that a principal must be granted access to the view V_1 in order to execute a given query. The formula $(V_1 \vee V_3)$ indicates that the principal must be granted access to *either* the view V_1 *or* the view V_3 . The formula $(V_1 \wedge V_6)$ indicates that the principal must be granted access to *both* the view V_1 *and* the view V_6 .

In general, a policy formula φ can be attached to any query Q (or more generally, any *set* of queries) over the dataset; the formula is then used by the reference monitor to decide whether a given principal can execute Q . Given the set of security views \mathbf{P} that the principal is granted access to, the reference monitor (RM) checks whether $\mathbf{P} \vdash \varphi$. If so, the RM will allow the principal to execute Q ; otherwise, the RM will deny the request.

This check can be performed in linear time by recursively applying the following rules:

$$\mathbf{P} \vdash 1 \text{ and } \mathbf{P} \neq 0$$

$$\mathbf{P} \vdash V \text{ if and only if } V \in \mathbf{P}$$

$$\mathbf{P} \vdash (\varphi \vee \psi) \text{ if and only if } \mathbf{P} \vdash \varphi \text{ or } \mathbf{P} \vdash \psi$$

$$\mathbf{P} \vdash (\varphi \wedge \psi) \text{ if and only if } \mathbf{P} \vdash \varphi \text{ and } \mathbf{P} \vdash \psi$$

It is important to realize that the check described above is agnostic to the question of how the views in \mathbf{P} are computed. Consequently, the model is relevant even for systems that rely on role-based access control or similar schemes where the process of computing \mathbf{P} can be arbitrarily complicated.

6.1.1 Generating Data-Derived Formulas

We now show how to generate data-derived policy formulas for database queries. Recall that a data-derived policy specifies that a principal with access to views in \mathbf{P} can receive the answer to query Q if and only if $\{Q\} \leq \mathbf{P}$. As explained in the Introduction, most policies used in practice are not data-derived; we believe this is because checking directly whether $\{Q\} \leq \mathbf{P}$ makes policy decisions computationally expensive and tricky for humans to understand.

On the other hand, given a policy formula φ , checking whether $\mathbf{P} \vdash \varphi$ is straightforward and can be performed quickly even for large formulas. We therefore propose to encode data-derived policy functions using policy formulas. That is, we want a system that can take a query Q and compute a policy formula φ such that, for any subset \mathbf{P} of the security views, $\mathbf{P} \vdash \varphi$ if and only if $\{Q\} \leq \mathbf{P}$. Intuitively, φ tells us precisely which combinations of the security views contain enough information to uniquely determine

the answer to Q . For example, consider the query

$$Q_7(h) :- \text{User}(1, n, h)$$

Given the security views defined in Figure 6.1, Q_7 can be answered if the principal is granted access to at least one of the views V_1 or V_3 . This is both a necessary and a sufficient condition. No other view except V_4 reveals the Hobby field of the User relation, and V_4 does not contain enough information to determine which hobby is associated with User 1. The policy formula for Q_7 is therefore

$$(V_1 \vee V_3)$$

The question arises whether it is always possible to compute the formula φ we desire. The answer is *yes*; there is a very deep connection between disclosure lattices and policy formulas. Given any set of queries \mathbf{Q} over the dataset there is a policy formula φ such that $\mathbf{P} \vdash \varphi$ if and only if $\mathbf{Q} \leq \mathbf{P}$ for every subset \mathbf{P} of the security views. Furthermore, it is possible to compute φ in a completely automated fashion.

Given a set of queries \mathbf{Q} , there is a simple algorithm that computes $\text{policy}(\mathbf{Q})$. The basic idea is to compute all the subsets of the security views which contain enough information to answer the queries in \mathbf{Q} . Formally, let \mathbf{V} denote the set of security views. Then we define

$$\text{policy}(\mathbf{Q}) = \bigvee_{\mathbf{P} \subseteq \mathbf{V}: \mathbf{Q} \leq \mathbf{P}} \bigwedge_{V \in \mathbf{P}} V$$

The outer disjunction iterates over all the subsets of \mathbf{P} that contain enough information to answer \mathbf{Q} . For each such subset, the conjunction of the views in \mathbf{P} suffices to answer \mathbf{Q} . We now verify the correctness of this algorithm.

Theorem 6.1.1. *For every set of queries $\mathbf{Q} \subseteq \mathbf{U}$ and every security policy $\mathbf{P} \subseteq \mathbf{V}$ it is the case that $\mathbf{Q} \leq \mathbf{P}$ if and only if $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$.*

Proof. First suppose that $\mathbf{Q} \leq \mathbf{P}$. Then $\bigwedge_{V \in \mathbf{P}} V$ appears as a disjunct in $\text{policy}(\mathbf{Q})$, and therefore $\mathbf{P} \vdash \bigwedge_{V \in \mathbf{P}} V$. It follows that $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$ as well.

For the other direction, suppose that $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$. Then there must be some $\mathbf{P}' \subseteq \mathbf{V}$ such that $\mathbf{P} \vdash \bigwedge_{V \in \mathbf{P}'} V$. This means that $\mathbf{P}' \subseteq \mathbf{P}$, and therefore $\mathbf{Q} \leq \mathbf{P}' \leq \mathbf{P}$. The desired result now follows from the transitivity of disclosure orders. \square

Efficiency, however, is another matter entirely. The number of security views can be quite large, and the size of a policy formula can be exponential in the number of security views. We must limit policy formulas to a manageable size if they are to be used in practice. We can ensure that this is the case by requiring the universe \mathbf{U} of possible views to be *decomposable*.

Formally, \mathbf{U} is said to be *decomposable* if for every query Q and all sets of views \mathbf{V} and \mathbf{V}' , $\{Q\} \leq \mathbf{V} \cup \mathbf{V}'$ implies $\{Q\} \leq \mathbf{V}$ or $\{Q\} \leq \mathbf{V}'$. Put another way, if neither \mathbf{V} alone nor \mathbf{V}' alone contains enough information to answer Q then the union of the two sets still will not contain enough information to answer Q .

Decomposability holds when we restrict our attention to the universe \mathbf{U}_{single} of single-atom conjunctive queries and views; it does not generally hold when applied to multi-atom queries and views in the obvious way. We restrict our attention to single-atom queries and views for now.

When decomposability holds, it is possible to compute concise policy formulas for arbitrary queries. Given a set of queries \mathbf{Q} and a set of security views \mathbf{V} , the following formula allows us to compute the formula governing \mathbf{Q} :

$$\text{policy}(\mathbf{Q}) = \bigwedge_{Q \in \mathbf{Q}} \bigvee_{V \in \mathbf{V}: \{Q\} \leq \{V\}} V$$

For any subset \mathbf{P} of the security views, $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$ if and only if $\mathbf{Q} \leq \mathbf{P}$. Consequently,

```

1: procedure POLICY(Q, V)
2:    $\varphi \leftarrow 1$ 
3:   for  $Q \in \mathbf{Q}$  do
4:      $\psi \leftarrow 0$ 
5:     for  $V \in \mathbf{V}$  do
6:       if  $\{Q\} \leq \{V\}$  then
7:          $\psi \leftarrow (\psi \vee V)$ 
8:       end if
9:     end for
10:     $\varphi \leftarrow (\varphi \wedge \psi)$ 
11:  end for
12:  return  $\varphi$ 
13: end procedure

```

Figure 6.2: Algorithm for computing policy formulas (decomposable case)

policy(**Q**) tells us exactly which combinations of the security views reveal enough information to answer all the queries in **Q**. Furthermore, its size is polynomial in both the number of queries and the number of security views. We now verify the correctness of this revised procedure.

Theorem 6.1.2. *Suppose that **U** is decomposable, and let $\mathbf{V} \subseteq \mathbf{U}$ be a set of security views. For every set of queries $\mathbf{Q} \subseteq \mathbf{U}$ and every policy $\mathbf{P} \subseteq \mathbf{V}$ we have $\mathbf{Q} \leq \mathbf{P}$ if and only if $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$.*

Proof. We first prove the Theorem for the special case where **Q** contains a single element Q . Since **U** is decomposable,

$$\{Q\} \leq \mathbf{P}$$

if and only if $\{Q\} \leq \{V\}$ for some $V \in \mathbf{P}$

if and only if $\{Q\} \leq \{V\}$ and $\mathbf{P} \vdash V$ for some $V \in \mathbf{V}$

if and only if $\mathbf{P} \vdash \varphi_Q$

$$\text{where } \varphi_Q = \bigvee_{V \in \mathbf{V}: \{Q\} \leq \{V\}} V$$

We now generalize the proof to sets \mathbf{Q} of arbitrary size:

$$\mathbf{Q} \leq \mathbf{P}$$

if and only if $\{Q\} \leq \mathbf{P}$ for every $Q \in \mathbf{Q}$

if and only if $\mathbf{P} \vdash \varphi_Q$ for every $Q \in \mathbf{Q}$

if and only if $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$ □

The algorithm in Figure 6.2 computes the policy formula for a set of queries \mathbf{Q} with respect to a collection of security views \mathbf{V} . It begins by setting the policy formula to 1 (Line 2) – an empty set of queries should always be permitted. It then loops through the list of queries in \mathbf{Q} (Lines 3-11), computes a separate policy formula for each query (Line 4-9), and returns the conjunction of all the resulting formulas (Line 12); the result is a policy formula over the views in \mathbf{V} .

The inner loop (Lines 4-9) computes the policy formula for a single query Q . It starts by setting the formula to 0 (Line 4) – if no security view contains enough information to answer Q then the query’s execution should be disallowed. It then loops through all the security views in \mathbf{V} and takes the disjunction of all the views that contain enough information to answer Q (Lines 5-9).

For example, let \mathbf{Q} contain the queries

$$Q_8(1, n) :- \text{User}(1, n, h)$$

$$Q_9(h) :- \text{User}(u, n, h)$$

and let \mathbf{V} consist of the views from Figure 6.1. Then

$$\text{policy}(\{Q_8\}) = V_1 \vee V_2 \vee V_3$$

$$\text{policy}(\{Q_9\}) = V_1 \vee V_4$$

and therefore

$$\text{policy}(\{Q_8, Q_9\}) = (V_1 \vee V_2 \vee V_3) \wedge (V_1 \vee V_4)$$

Given a set of queries \mathbf{Q} , we have shown that there is a formula $\text{policy}(\mathbf{Q})$ such that $\mathbf{Q} \leq \mathbf{P}$ if and only if $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$. It is natural to ask whether we can go in the other direction: given a formula φ , can we find a set of queries \mathbf{Q} such that $\mathbf{Q} \leq \mathbf{P}$ if and only if $\mathbf{P} \vdash \varphi$? In general, the answer is *no*. There are two basic difficulties. The first is that φ may be inconsistent. For instance, it is possible to grant a principal access to V_1 but not V_2 even though the former reveals strictly more information about the dataset than the latter. Even if we disallow such inconsistencies, however, we cannot guarantee that a policy will correspond to any meaningful level of information disclosure. For instance $(V_1 \vee V_5)$ is a policy that requires access to *either* the `User` relation *or* the `Friend` relation (but does not require access to both). Unless we permit foreign-key constraints, the two views reveal completely disjoint types of information about the dataset, and so only a trivial (database-independent) query can be answered using the information that is common to the two views.

In summary, we have shown how to represent the security policy governing any query over the dataset as a data-derived policy formula. We do this by exploiting a deep connection between disclosure lattices that precisely quantify information disclosure and policy formulas that are easy for a reference monitor to evaluate. For every query Q we compute a formula φ ; the reference monitor makes policy decisions by evaluating φ against the list of permissions \mathbf{P} that were granted to the principal. If the universe of queries is decomposable then policy formulas are guaranteed to be small and easy to reason about. However, it is possible to compute policy formulas for queries even when decomposability does not hold.

6.1.2 Minimal Policy Formulas

In the discussion above, our goal was to characterize *all* the sets of views that contained enough information to uniquely determine the answer to a given query. However, not all sets of views are created equal. Consider the policy formula

$$\text{policy}(\{Q_8, Q_9\}) = (V_1 \vee V_2 \vee V_3) \wedge (V_1 \vee V_4)$$

from our running example. A principal who wishes to learn the answers to Q_8 and Q_9 can either request access to V_1 or can request access to V_2 and V_4 . The latter is preferable because it reveals less information about the dataset: a principal who is granted access to V_1 can see which hobbies are associated with which users, whereas a principal who is granted access to V_2 and V_4 generally cannot. We now consider the problem of characterizing all the sets of views that contain enough information to answer Q but are *minimal* in the sense that they reveal as little additional information about the dataset as possible. (As we will see, it is also minimal in the usual set-theoretic sense.)

Given a set of queries \mathbf{Q} and a set of security views \mathbf{V} , we begin by computing the policy formula for \mathbf{Q} . We then apply the Distributive Law repeatedly until the formula is in Disjunctive Normal Form. For example, when we apply the Distributive Law to the policy formula above we obtain the expanded formula

$$(V_1 \wedge V_1) \vee (V_1 \wedge V_4) \vee (V_2 \wedge V_1) \vee (V_2 \wedge V_4) \vee (V_3 \wedge V_1) \vee (V_3 \wedge V_4)$$

We next create a new set of axioms \mathcal{A} that encodes the relative amounts of information disclosed by different sets of views. Let V be a security view in \mathbf{V} , and let $\mathbf{V}' \subseteq \mathbf{V}$. If $\{V\} \leq \mathbf{V}'$ then \mathcal{A} contains the implication

$$\left(\bigwedge \mathbf{V}'\right) \Rightarrow V$$

$$\begin{array}{ll} V_1 \Rightarrow V_2 & V_1 \Rightarrow V_4 \\ V_1 \Rightarrow V_3 & V_5 \Rightarrow V_6 \end{array}$$

Figure 6.3: Axioms encoding relative information revealed by sets of views.

In our running example, \mathcal{A} contains the axiom $(V_1 \Rightarrow V_2)$ because $\{V_2\} \leq \{V_1\}$. Intuitively, this axiom states that any principal who is granted access to V_1 should also be granted access to V_2 because V_2 reveals strictly less information about the dataset than V_1 . A full list of axioms in our running example is shown in Figure 6.3.

In general, the size of \mathcal{A} can be exponential in the number of security views. However, if the universe of queries is decomposable then \mathcal{A} only needs to contain axioms of the form $(V' \Rightarrow V)$ where V and V' are individual views. In this case, the size of \mathcal{A} is at most quadratic in the number of security views. To justify this optimization, suppose that $\{V\} \leq \{V', V''\}$. Then $\{V\} \leq \{V'\}$ or else $\{V\} \leq \{V''\}$. In the first case we have $\mathcal{A} \vdash V' \Rightarrow V$ and in the second we have $\mathcal{A} \vdash V'' \Rightarrow V$. In either case it follows that $\mathcal{A} \vdash (V' \wedge V'') \Rightarrow V$.

Finally, if $policy(\mathbf{Q})$ contains a pair of disjuncts φ and φ' such that $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ then we remove the disjunct φ from $policy(\mathbf{Q})$. In the example above,

$$\mathcal{A} \vdash (V_1 \wedge V_1) \Rightarrow (V_2 \wedge V_1)$$

so we remove the disjunct $(V_1 \wedge V_1)$ from $policy(\mathbf{Q})$. We repeat the process until no more disjuncts can be removed. This step eliminates any *non-minimal* disjuncts in $policy(\mathbf{Q})$. After applying the procedure to the policy formula in our running example, every disjunct that references V_1 is removed, and we are left with the formula

$$(V_2 \wedge V_4) \vee (V_3 \wedge V_4)$$

The result $minpolicy(\mathbf{Q})$ of the procedure described above is logically equivalent to the original policy formula $policy(\mathbf{Q})$. For any policy \mathbf{P} we have $\mathcal{A} \cup \mathbf{P} \vdash policy(\mathbf{Q})$ if

and only if $\mathcal{A} \cup \mathbf{P} \vdash \text{minpolicy}(\mathbf{Q})$. This is true because the procedure outlined above first converts the policy formula to DNF and then removes disjuncts φ such that $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$. Converting a formula to DNF does not affect its truth value. Furthermore, if $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ then any policy \mathbf{P} such that $\mathcal{A} \cup \mathbf{P} \vdash \varphi$ also satisfies $\mathcal{A} \cup \mathbf{P} \vdash \varphi'$, so removing φ won't affect the truth value of $\text{minpolicy}(\mathbf{Q})$.

The resulting policy formula is also *minimal* in two different senses of the word. First, any disjuncts that revealed more information about the dataset than we need to answer \mathbf{Q} have been removed: if $\mathbf{V} \leq \mathbf{V}'$ and $\mathbf{V}' \leq \mathbf{V}$ then the disjunct associated with \mathbf{V}' will imply the disjunct associated with \mathbf{V} , and will therefore be removed. And second, removing any more disjuncts from $\text{minpolicy}(\mathbf{Q})$ would change its truth value. Let φ be a disjunct of $\text{minpolicy}(\mathbf{Q})$, and let $\text{badpolicy}(\mathbf{Q})$ be the formula obtained by removing φ from $\text{minpolicy}(\mathbf{Q})$. Our claim is that $\text{badpolicy}(\mathbf{Q})$ and $\text{minpolicy}(\mathbf{Q})$ can't be logically equivalent. Let \mathbf{P} be a policy that contains exactly the views that appear in φ . Then $\mathcal{A} \cup \mathbf{P} \vdash \text{minpolicy}(\mathbf{Q})$. On the other hand, $\mathcal{A} \cup \mathbf{P} \not\vdash \varphi'$ for every disjunct φ' in $\text{badpolicy}(\mathbf{Q})$, so that $\mathcal{A} \cup \mathbf{P} \not\vdash \text{badpolicy}(\mathbf{Q})$.

The discussion above demonstrates how to compute the *minimal* sets of views that contain enough information to answer a principal's queries but which reveal as little additional information about the dataset as possible. Minimal policy formulas are closely related to disclosure labels: a minimal policy formula identifies *all* the minimal subsets of the security views that determine a query's answer, whereas a disclosure label ensures that the security views are selected so that the minimal set of views is essentially *unique*.

Unfortunately, the process outlined above requires us to expand a CNF formula to DNF, which can result in exponential blowup. We therefore believe that the algorithm discussed above is primarily of theoretical rather than practical interest, at least in its current form.

6.2 Explanations

Policy formulas are not just useful for enforcement; they can be used to generate *explanations*, which are mathematical objects that justify the RM's policy decisions.

Explanations come in two varieties: *why-so explanations* and *why-not explanations*. If the execution of a query Q is authorized, a why-so explanation indicates which of the principal's permissions were responsible for the positive authorization decision. If the execution of Q is not authorized, a why-not explanation indicates which additional permissions need to be granted before the query can be successfully executed. Both types of explanations can be efficiently computed from $\text{policy}(Q)$.

As before, let \mathbf{Q} denote a set of queries, \mathbf{V} denote a set of security views, and let $\mathbf{P} \subseteq \mathbf{V}$ be the subset of the security views that a principal has been granted access to. A why-so explanation for \mathbf{Q} characterizes exactly which subsets of \mathbf{P} contain enough information to answer \mathbf{Q} . It can be obtained by replacing every view V in $\text{policy}(\mathbf{Q})$ that the principal has not been granted access to with the constant 0. For example, suppose $\mathbf{Q} = \{Q_8, Q_9\}$, so that

$$\text{policy}(\mathbf{Q}) = (V_1 \vee V_2 \vee V_3) \wedge (V_1 \vee V_4)$$

If the principal is granted access to V_1 and V_2 then the why-so explanation for \mathbf{Q} w.r.t. $\{V_1, V_2\}$ is

$$(V_1 \vee V_2 \vee 0) \wedge (V_1 \vee 0) = (V_1 \vee V_2) \wedge (V_1) = V_1$$

indicating that V_1 was solely responsible for the positive authorization decision. If the principal is instead granted access to $\{V_2, V_3, V_4\}$, the why-so explanation for \mathbf{Q} is

$$(0 \vee V_2 \vee V_3) \wedge (0 \vee V_4) = (V_2 \vee V_3) \wedge (V_4)$$

indicating that in order to answer the query, the principal would need access to V_4 and at least one of V_2 and V_3 .

We can now be ready to formally define why-so explanations. Intuitively, the goal is to characterize the subsets \mathbf{P}' of the current policy \mathbf{P} that satisfy φ .

Definition 6.2.1. A why-so explanation for φ w.r.t. a policy \mathbf{P} is a formula φ_S such that, for every $\mathbf{P}' \subseteq \mathbf{P}$, we have $\mathbf{P}' \vdash \varphi_S$ if and only if $\mathbf{P}' \vdash \varphi$. Furthermore, φ_S is only permitted to reference the views in \mathbf{P} .

We now verify the correctness of the procedure outlined above. Given a formula φ , we write $\varphi\left(\frac{0}{\mathbf{V}\setminus\mathbf{P}}\right)$ to denote the formula obtained by taking φ and replacing each view in $(\mathbf{V}\setminus\mathbf{P})$ with the constant 0.

Theorem 6.2.2. $\varphi\left(\frac{0}{\mathbf{V}\setminus\mathbf{P}}\right)$ is a why-so explanation for φ w.r.t. \mathbf{P} .

Proof. The proof is by structural induction on φ . Fix $\mathbf{P}' \subseteq \mathbf{P}$.

- If $\varphi = V$ then

$$\mathbf{P}' \vdash \varphi\left(\frac{0}{\mathbf{V}\setminus\mathbf{P}}\right)$$

if and only if $V \notin (\mathbf{V}\setminus\mathbf{P})$ and $V \in \mathbf{P}'$

if and only if $V \in \mathbf{P}$ and $V \in \mathbf{P}'$

if and only if $V \in \mathbf{P}'$ (since $\mathbf{P}' \subseteq \mathbf{P}$)

if and only if $\mathbf{P}' \vdash \varphi$

- $\varphi = (\psi_1 \vee \psi_2)$ then

$$\mathbf{P}' \vdash \varphi\left(\frac{0}{\mathbf{V}\setminus\mathbf{P}}\right)$$

if and only if $\mathbf{P}' \vdash \psi_1\left(\frac{0}{\mathbf{V}\setminus\mathbf{P}}\right)$ or $\mathbf{P}' \vdash \psi_2\left(\frac{0}{\mathbf{V}\setminus\mathbf{P}}\right)$

if and only if $\mathbf{P}' \vdash \psi_1$ or $\mathbf{P}' \vdash \psi_2$ (by induction)

if and only if $\mathbf{P}' \vdash \varphi$

- $\varphi = (\psi_1 \wedge \psi_2)$ then

$$\begin{aligned} & \mathbf{P}' \vdash \varphi \left(\frac{0}{\mathbf{V} \setminus \mathbf{P}} \right) \\ & \text{if and only if } \mathbf{P}' \vdash \psi_1 \left(\frac{0}{\mathbf{V} \setminus \mathbf{P}} \right) \text{ and } \mathbf{P}' \vdash \psi_2 \left(\frac{0}{\mathbf{V} \setminus \mathbf{P}} \right) \\ & \text{if and only if } \mathbf{P}' \vdash \psi_1 \text{ and } \mathbf{P}' \vdash \psi_2 \text{ (by induction)} \\ & \text{if and only if } \mathbf{P}' \vdash \varphi \quad \square \end{aligned}$$

If the RM's authorization decision is negative, a why-not explanation identifies the additional permissions that would need to be granted in order to reverse the decision. A why-not explanation is obtained by replacing every view V in $\text{policy}(\mathbf{Q})$ that the principal *has* been granted access to with the constant 1. If

$$\text{policy}(\mathbf{Q}) = (V_1 \vee V_2 \vee V_3) \wedge (V_1 \vee V_4)$$

then the why-not explanation for \mathbf{Q} w.r.t. $\{V_2, V_3\}$ is

$$(V_1 \vee 1 \vee 1) \wedge (V_1 \vee V_4) = 1 \wedge (V_1 \vee V_4) = (V_1 \vee V_4)$$

indicating that in order to execute the queries in \mathbf{Q} , the principal would need to be granted additional access to at least one of the permissions V_1 and V_4 . Similarly, the why-not explanation for \mathbf{Q} w.r.t. $\{V_4\}$ is

$$\begin{aligned} & (V_1 \vee V_2 \vee V_3) \wedge (V_1 \vee 1) \\ & = (V_1 \vee V_2 \vee V_3) \wedge 1 \\ & = (V_1 \vee V_2 \vee V_3) \end{aligned}$$

indicating that the principal would need to be granted access to at least one of the views V_1 , V_2 , and V_3 before all the queries in \mathbf{Q} could be answered.

We next formalize why-not explanations. The basic idea is to characterize the *additional views* \mathbf{P}'' that would need to be added to \mathbf{P} in order to ensure that φ is satisfied.

Definition 6.2.3. A why-not explanation for φ w.r.t. a policy \mathbf{P} is a formula φ_N such that, for every $\mathbf{P}' \subseteq (\mathbf{V} \setminus \mathbf{P})$, we have $\mathbf{P}' \vdash \varphi_N$ if and only if $\mathbf{P} \cup \mathbf{P}' \vdash \varphi$. Furthermore, φ_N is only permitted to reference the views in $(\mathbf{V} \setminus \mathbf{P})$.

We now verify the correctness of the procedure outlined above. As before, $\varphi\left(\frac{1}{\mathbf{P}}\right)$ denotes the formula obtained taking φ and replacing every view in \mathbf{P} with the constant 1.

Theorem 6.2.4. $\varphi\left(\frac{1}{\mathbf{P}}\right)$ is a why-not explanation for φ w.r.t. \mathbf{P} .

Proof. The proof is by structural induction on φ . Fix $\mathbf{P}' \subseteq (\mathbf{V} \setminus \mathbf{P})$.

- If $\varphi = V$ then

$$\mathbf{P}' \vdash \varphi\left(\frac{1}{\mathbf{P}}\right)$$

if and only if $V \in \mathbf{P}$ or $V \in \mathbf{P}'$

if and only if $V \in \mathbf{P} \cup \mathbf{P}'$

if and only if $\mathbf{P} \cup \mathbf{P}' \vdash \varphi$

- If $\varphi = (\psi_1 \vee \psi_2)$ then

$$\mathbf{P}' \vdash \varphi\left(\frac{1}{\mathbf{P}}\right)$$

if and only if $\mathbf{P}' \vdash \psi_1\left(\frac{1}{\mathbf{P}}\right)$ or $\mathbf{P}' \vdash \psi_2\left(\frac{1}{\mathbf{P}}\right)$

if and only if $\mathbf{P} \cup \mathbf{P}' \vdash \psi_1$ or $\mathbf{P} \cup \mathbf{P}' \vdash \psi_2$ (by induction)

if and only if $\mathbf{P} \cup \mathbf{P}' \vdash \psi_1 \vee \psi_2$

- If $\varphi = (\psi_1 \wedge \psi_2)$ then

$$\begin{aligned} & \mathbf{P}' \vdash \varphi \left(\frac{1}{\mathbf{P}} \right) \\ & \text{if and only if } \mathbf{P}' \vdash \psi_1 \left(\frac{1}{\mathbf{P}} \right) \text{ and } \mathbf{P}' \vdash \psi_2 \left(\frac{1}{\mathbf{P}} \right) \\ & \text{if and only if } \mathbf{P} \cup \mathbf{P}' \vdash \psi_1 \text{ and } \mathbf{P} \cup \mathbf{P}' \vdash \psi_2 \text{ (by induction)} \\ & \text{if and only if } \mathbf{P} \cup \mathbf{P}' \vdash \psi_1 \wedge \psi_2 \quad \square \end{aligned}$$

Why-so and why-not explanations are helpful to both the principal and the system administrator. From the administrator's perspective, if the principal holds some permissions that never show up in why-so explanations for his or her queries, this may be an indication that the principal is *overprivileged* and that those unused permissions should be revoked. On the other hand, from the principal's perspective, if a query is denied, the why-not explanation provides clear guidance on which additional permissions to request from the appropriate granting entity. With our data-derived policy formulas, both kinds of explanations are straightforward to compute at query time.

6.3 Query Pricing

In addition to assisting in the derivation of why-so and why-not explanations, explicit policy expressions can also be applied to the problem of *query pricing*. [35] Adapted to the terminology of our framework, an administrator defines a set of security views \mathbf{V} and independently assigns a separate price $\pi(V)$ to each view $V \in \mathbf{V}$. A user who purchases a subset \mathbf{P} of the security views can issue any query he likes, so long as we can ensure that the query's answer is uniquely determined by the answers to the views he has purchased.

Given a set of queries \mathbf{Q} that a principal wants answered, our goal is to find a subset

$\mathbf{P} \subseteq \mathbf{V}$ such that $\mathbf{Q} \leq \mathbf{P}$ and the total price $\sum_{V \in \mathbf{Q}} \pi(V)$ is minimal subject to this restriction. The first condition ensures that \mathbf{P} contains enough information to answer all the queries in \mathbf{Q} , while the second ensures that the total purchase price is minimized.

The problem can be solved by setting up a relatively small 0 – 1 linear program. We associate a distinct variable with P_V with every security view $V \in \mathbf{V}$. The variable will be set to 1 if the corresponding security view is purchased and will be set to 0 otherwise. Our goal is to minimize the total purchase price

$$\sum_{V \in \mathbf{V}} \pi(V) \cdot P_V$$

subject to the constraint that $S \geq 1$, where S is obtained by replacing every view V in $\text{policy}(\mathbf{Q})$ with the corresponding variable P_V , replacing conjunction with multiplication, and replacing disjunction with addition. This ensures that the set of purchased views contains enough information to uniquely determine the answers to all the queries in \mathbf{Q} . In our running example, suppose we assign prices to the security views as follows:

$$\begin{array}{lll} \pi(V_1) = 1 & \pi(V_2) = 1/2 & \pi(V_3) = 1/3 \\ \pi(V_4) = 1/4 & \pi(V_5) = 1/5 & \pi(V_6) = 1/6 \end{array}$$

Then the goal is to minimize the total price

$$1 \cdot P_{V_1} + (1/2) \cdot P_{V_2} + (1/3) \cdot P_{V_3} + (1/4) \cdot P_{V_4} + (1/5) \cdot P_{V_5} + (1/6) \cdot P_{V_6}$$

subject to the constraint that

$$(P_{V_1} + P_{V_2} + P_{V_3}) \cdot (P_{V_1} + P_{V_4}) \geq 1$$

Any variable assignment that satisfies this constraint will induce a solution to

$$(V_1 \vee V_2 \vee V_3) \wedge (V_1 \vee V_4)$$

which is the policy formula for \mathbf{Q} , and vice versa. Hence, S ensures that the purchased views will contain enough information to answer all the queries in \mathbf{Q} .

The disclosure orders that we focus on are all *dataset-independent*. Hence, our solution yields a *dataset-independent* version query pricing, whereas previous work by Koutris et al. (e.g., [35]) has focused on the problem *dataset-dependent* query pricing. The main advantages of using a dataset-independent model are that query prices can never leak valuable information about the contents of the dataset and the computational overhead for query pricing tends to be much lower (since our algorithms are independent of the contents of the dataset). In addition, we can deal with security views that contain column projections, which are difficult to handle in the dataset-dependent model. On the other hand, dataset-dependent query pricing can sometimes exploit information about the current state of the dataset to generate solutions that are cheaper than the best dataset-independent alternatives.

CHAPTER 7

EXPLAINABLE SECURITY IN PRACTICE

We now discuss a prototype system that we built to evaluate the feasibility of computing policy formulas and explanations for SQL queries. Our prototype system combines the algorithms developed to handle complex queries and policy constraints from Chapter 5 with the explainable security model discussed in Chapter 6.

The high-level architecture of our system is shown in Figure 7.1. Computing policy formulas is a two-stage process. First, queries issued by a principal are fed to a SQL processor that *generalizes* them to filter-project queries. Our generalizer is architecturally similar to a compiler, but the translation process is conservative: the generalized versions of the queries disclose at least as much information as the original SQL queries, and may reveal strictly more information in some cases. As discussed in Chapter 5, this may cause the reference monitor to overestimate the information disclosed by and consequently reject certain legal queries. It will, however, ensure that illegal queries are never accepted. A policy formula therefore represents an upper bound on the information needed to answer a query.

Security views are similarly translated into filter-project queries at initialization time; the translation process for security views is lossless due to the restriction we imposed above.¹ Once the translation is complete, the system computes a policy formula for the generalized query using the algorithm from Figure 6.2. The entire pipeline is database-independent; policy formulas and explanations can reveal information about the database schema but cannot leak information about the contents of the underlying database.

The remainder of this Chapter describes the architecture of our system in more detail.

¹The semantics of equality comparisons involving NULL values are different for SQL and filter-project queries. Consequently, security views must only contain comparisons between pairs of non-NULL values.

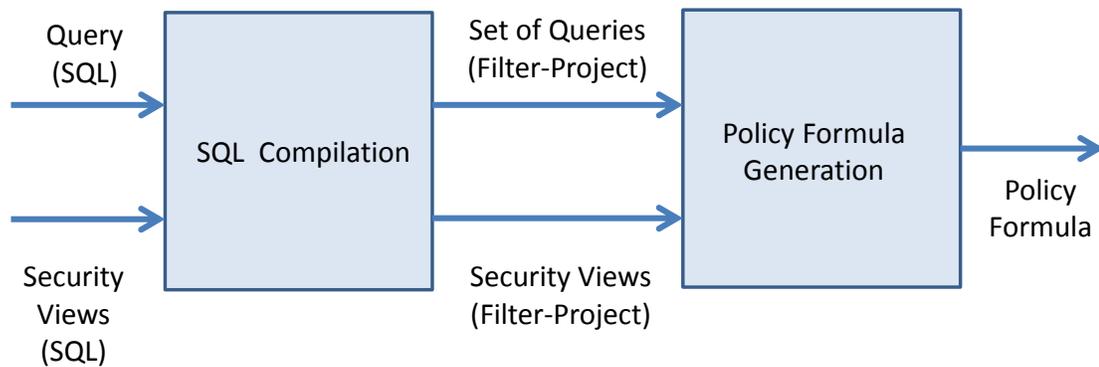


Figure 7.1: Computing policy formulas – System Architecture

In Section 7.1 we discuss the compilation pipeline used by our system. Then in Section 7.2 we discuss an experimental evaluation of our system.

7.1 Generalization Pipeline

Our generalizer begins by lexing and parsing the raw SQL queries using JSqlParser [1], an open-source query parsing library written in Java. We then execute a series of passes on the resulting *abstract syntax tree* (AST). Each pass corresponds to a complete traversal of the AST. A complete depiction of the passes is seen in Figure 7.2.

Our running example focuses on the query

```

SELECT name FROM User U1, Friend F1
WHERE uid1 = 1 AND uid2 = uid
  
```

Preprocessing and column resolution: Parsing the raw SQL queries yields an AST representation as mentioned earlier. Once the AST is available, we identify table instances and perform column resolution. In our example, we identify U1 as a table instance associated with the User table and F1 as a table instance associated with the Friend table. Using information about the database schema, we infer that the columns

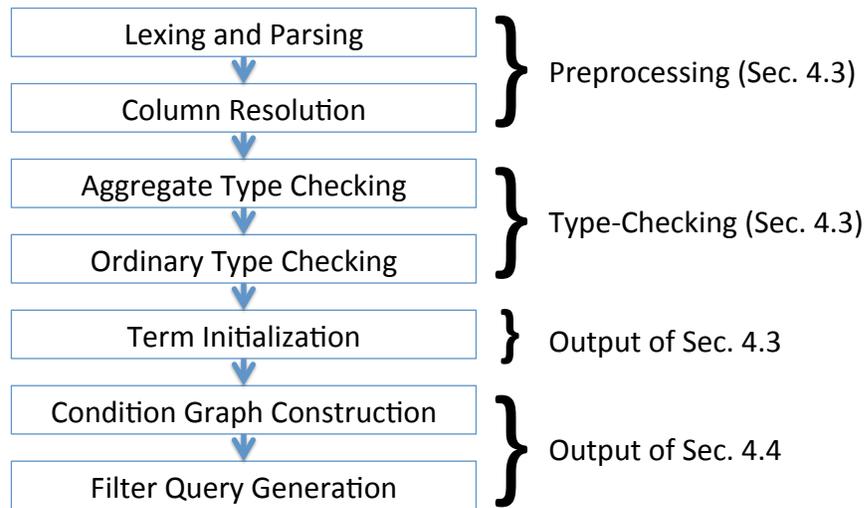


Figure 7.2: Query generalization pipeline

uid and name belong to U1 whereas the columns uid1 and uid2 belong to F1. When applicable, column resolution also takes into account information about the query’s nested structure.

Type-checking: We perform aggressive type-checking on the AST in order to ensure that it corresponds to a semantically well-defined query. The motivation for performing type-checking is simple: we can’t reason about the disclosure of a query unless we understand its semantics. Our system performs two different kinds of type-checking. *Aggregate type-checking* detects bad queries such as

```
SELECT SUM(SUM(U1.uid)) FROM User U1
```

that mix aggregate and non-aggregate expressions in illegal ways. *Ordinary type-checking* detects bad queries such as

```
SELECT * FROM User U1 WHERE U1.uid IN
(SELECT F1.uid1, F1.uid2 FROM Friend F1)
```

that provide invalid inputs to built-in functions, operators, and predicates.

Term initialization: The next step is to convert the AST for a SQL query Q into a collection of projection queries Q_1, Q_2, \dots, Q_n that together reveal enough information to uniquely determine the answer to Q on any possible dataset.

The extraction proceeds as follows. First, we generate one query for each table instance in Q . For example, given any of the SQL queries from Figure 5.6a, Figure 5.6b, or Figure 5.6c, we generate two queries: Q_{U1} and Q_{F1} . The sole body atom of Q_{U1} references the `User` relation whereas the body atom of Q_{F1} references the `Friend` relation.

We next determine which variables in our queries should be existential and which should be distinguished. A variable is marked as existential if the corresponding column is never referenced in the definition of Q , and is marked as distinguished in all other cases. This ensures that the output queries reveal all the columns needed to evaluate any arithmetic expression or inequality or logical predicate that appears in Q . We obtain the same values for Q_{U1} and Q_{F1} for all of the queries in Figures 5.6a, 5.6b, and 5.6c:

$$Q_{U1}(u, n) :- U(u, n, h)$$

$$Q_{F1}(u_1, u_2) :- F(u_1, u_2)$$

If we were to stop the generalization process at this point we would have a collection of single-atom conjunctive queries. However, our ultimate goal is to generate a collection of filter-project queries that can keep track of interactions between different table instances in a query. This is done by constructing and then traversing a *condition graph* using the algorithms discussed in Section 5.4. After performing this step, the updated versions of Q_{U1} and Q_{U2} will be defined as follows:

$$Q_{U1}(u, n) :- U(u, n, h) \bowtie F(1, u) \wedge (u_1 = 1) \wedge (u_2 = u)$$

$$Q_{F1}(1, u) :- F(1, u) \bowtie U(u, n, h)$$

These two filter-project queries together comprise the final output of the generalizer. We can then apply the algorithms from Chapter 6 to generate policy formulas and explanations.

7.2 Experimental Evaluation

We now present an evaluation of our prototype system. The goal of our experimental evaluation was three-fold. First, we wanted to verify that SQL queries containing a wide range of commonly used features could be correctly handled by the generalization pipeline discussed in Section 7.1. Second, we wanted to determine whether the language of filter-project queries described in Section 5.1 was powerful enough to represent a variety of practical security constraints. And third, we wanted to determine whether policy formulas could be generated quickly enough to be used in practical systems.

We implemented a prototype system in Java. The prototype's architecture is depicted in Figure 7.3. Our system consisted of three main components: (i) a generalizer for translating SQL queries into filter-project queries, (ii) a module for representing and reasoning about filter-project queries, and (iii) a module for representing and evaluating policy formulas and generating why-so and why-not explanations. The implementation of our generalizer consisted of just over 5,500 lines of Java code, while filter-project queries were implemented in about 1,000 lines, and policy formulas were implemented in 250.

For the first goal, we began by writing end-to-end tests for our generalizer based on several dozen SQL queries taken from a standard undergraduate database textbook. (See Chapter 5 of [45].) We gradually added more tests over time to keep track of the border cases we encountered. At the time of publication, we had 110 of our 115 end-to-end

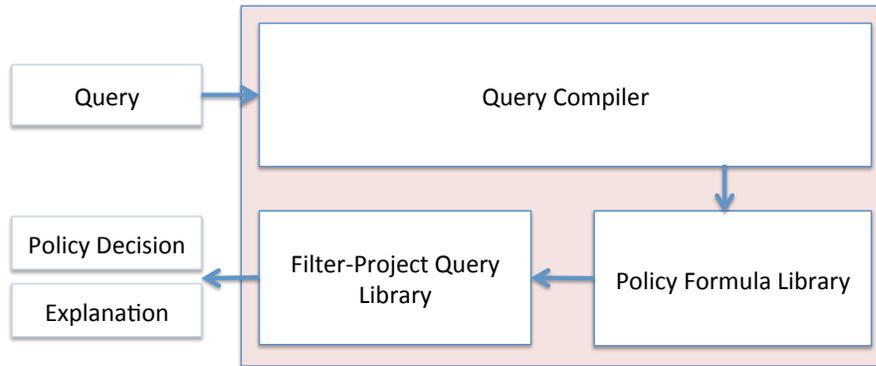


Figure 7.3: Architecture of Prototype System

tests running successfully, and the remaining five were rejected as being semantically invalid. Of these, three were rejected due to limitations of JSqlParser. The last two were valid according to the SQL standard, but were considered to be invalid by both MySQL and PostgreSQL. The results gave us confidence that our compilation-based approach to disclosure control could be successfully applied to a broad range of real-world SQL features.

For the second and third goals, we ran experiments on a database inspired by the one used by the Facebook Query Language (FQL). [4] The schema has a rich and diverse set of security requirements. For example, albums containing media such as photos and videos can be made visible to friends of the current user, to friends of friends, or to the public at large. The membership of a group can be made world-visible, can be restricted to members of that group, or can be completely hidden. Lists of attendees for events can similarly be made world-visible or can be restricted to other attendees. All of these contingencies were accounted for using filter-project security views.

Our test database schema contained 18 relations with a grand total of 269 distinct columns – or an average of 15 columns per relation. We deliberately selected relations that had nontrivial security policies. The number of columns per relation ranged from two for

the `Friend` relation (which stores information about friendship relations between users) to 62 for the `Application` table (which stores information about third-party Facebook Apps). We ended up defining a total of 75 security views. Equality constraints appeared in all but one of our security views; the exception was a view over the `Application` table that listed certain types of publicly available information about Facebook Apps. Filter atoms appeared in 59 security views. Column projections were used in just four of the security views.

A useful but unexpected consequence of the restrictions we impose on our handling of *filter-project* queries was that it is possible to define a form of inheritance for cases where the visibility of rows in one table depends on the visibility of rows in a different table. For instance, access to a `photo` is determined by the `album` which contains that photo. Any given album can be made world-visible or else its visibility can be restricted to the owner's friends or to friends of friends. We defined multiple security views over the `album` relation to handle the latter contingencies. However, we only needed to define one security view over the `photo` relation:

```
SELECT * FROM photo
WHERE aid IN (SELECT aid FROM album);
```

In effect, the security view above forces any principal who queries the `photo` relation to perform a join on `aid` (Album ID) with the corresponding row of the `album` relation. Since our system performs a separate policy check for every table instance of the input query, this means that a principal can access information about a `photo` only if he or she is granted access to a view containing information about the corresponding `album`. In this way, the visibility of `photos` is inherited from the visibility of the `albums` to which they belonged.

In addition to the qualitative experiments above, we also evaluated the performance

of our prototype system. Our experiments were run on a workstation with two 4-core 2.13 GHz Intel Xeon processors and 50 GB of RAM. Our code was run on the OpenJDK 1.7 VM on Ubuntu 12.04.

Our experiments made use of a synthetic query generator that worked by recursively stringing together predefined templates. Our use of templates allowed us to generate realistic-looking queries whose joins and equality predicates encoded meaningful security-related constraints. Two sample queries generated by our system (chosen completely at random) are shown below:

```
SELECT R1.owner, R1.video_count FROM album AS R1
WHERE R1.visible = 'friends-of-friends' AND EXISTS
(SELECT DISTINCT 1 FROM friend AS R2, friend AS R3
WHERE R2.uid1 = 4 AND R2.uid2 = R3.uid1
AND R3.uid2 = 4);
```

```
SELECT R1.size, R1.src, R1.width
FROM photo_src AS R1 WHERE EXISTS
(SELECT DISTINCT 1 FROM photo AS R2, album AS R3
WHERE R1.photo_id = R2.pid
AND R2.aid = R3.aid
AND R3.visible = 'everyone');
```

Each of the queries we generated contained between one and six table instances; the median was 2 and the mean was 2.5.

We measured the wall time needed to (i) randomly generate queries, (ii) lex and parse them, (iii) extract filter-project queries from the parsed output, (iv) compute policy formulas from the resulting filter-project queries, and (v) generate a policy decision for each query, a why-so explanation for each permitted query, and a why-not explanation for each denied query. Since each stage depended on the output of the previous stage, these

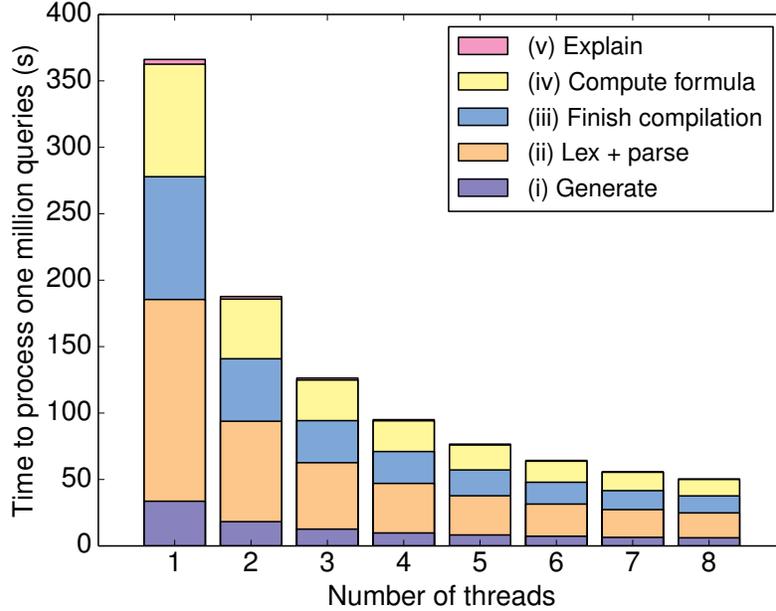


Figure 7.4: Performance for ordinary queries

measurements were cumulative: we measured the time for (i) alone, the time for (i) + (ii), and so on. We also varied the number of execution threads between 1 and 8. All of our algorithms are trivially parallelizable, and the threads operated completely independently of each other. In each case, we measured the time needed to process 1,000,000 queries. All numbers are averaged across five runs; the variation across runs was less than one second in almost all cases.

The results are shown in Figure 7.4. As expected, throughput scaled nearly linearly with the number of cores. The analysis pipeline took a total of 366 seconds on a single core or 50 seconds on 8 cores. If we exclude the time needed to randomly generate queries, these numbers change to 333 and 44 seconds respectively. If we focus on the last three stages (which are where we claim algorithmic contributions), the numbers fall to 181 and 26 seconds respectively. All computation was CPU-bound, and memory consumption ranged from 60 MB to 220 MB per thread. The time needed to perform

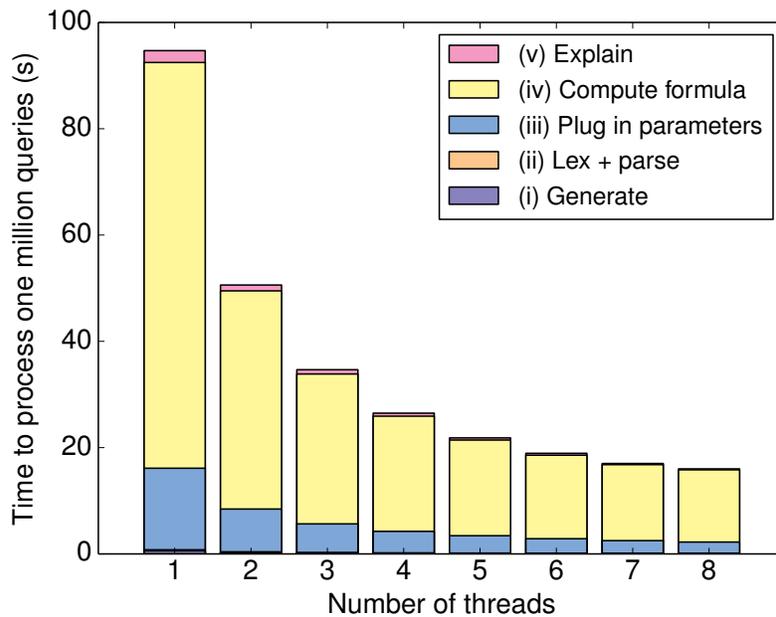


Figure 7.5: Performance for prepared statements

policy checks and generate explanations was negligible compared to the rest of the system – about 3.5 seconds on a single core or 0.5 seconds on eight cores – and is barely visible in Figure 7.4.

Since query compilation time dominated the benchmark numbers in Figure 7.4, we speculated that throughput could be significantly improved by using prepared statements. The filter-project queries associated with prepared statements only needed to be calculated once (as a preprocessing step), and could be retrieved on demand when the prepared statements were executed. Parameters passed to prepared statements at execution time could be substituted directly into the generalized filter-project queries, bypassing the need for a heavy-weight compilation step.

Results are shown in Figure 7.5. Our use of prepared statements yielded about a 3x improvement in the system’s overall throughput. The time spent on query generation, lexing and parsing was negligible, and the time needed to retrieve filter-project queries

and plug in parameter values paled in comparison with the time needed to perform end-to-end query compilation. As a result, computation of policy formulas dominated the running time.

The experiments above focus on the generalizer’s performance for semijoin queries. These types of queries effectively stress steps (iv) and (v) of our query analysis pipeline because they tend to yield large filter-project queries that have many filter atoms. Our final experiment was designed to get a better idea of the generalizer’s performance on real SQL queries. Our workload consisted of queries drawn from a list of 86 predefined strings, most of which were taken from Chapter 5 of [45]. The workload incorporated a diverse set of features, including aggregates, arithmetic comparisons, `GROUP BY` statements and aggregates, inner, outer, and semijoins, set operations, and temporary tables. The security constraints that naturally arose for our sample database were simpler than for our Facebook case study. Because the third experiment focused on the compilation phase of our pipeline, we defined only nine security views: three views for each of three base relations.

Results for the third experiment are shown in Figure 7.6. Query generation time was essentially zero because the queries were selected from a predefined list. The total time needed to generalize 1,000,000 queries (which was the main focus of our experiment) was around 111 seconds on a single core, or around 15 seconds when distributed across eight cores. The number is significantly lower than in earlier experiments, where compilation time averaged 244 seconds on a single core or 31 seconds on eight cores. This is not surprising because queries in the synthetically generated workload from our previous experiment tended to yield relatively large ASTs and well-connected condition graphs.

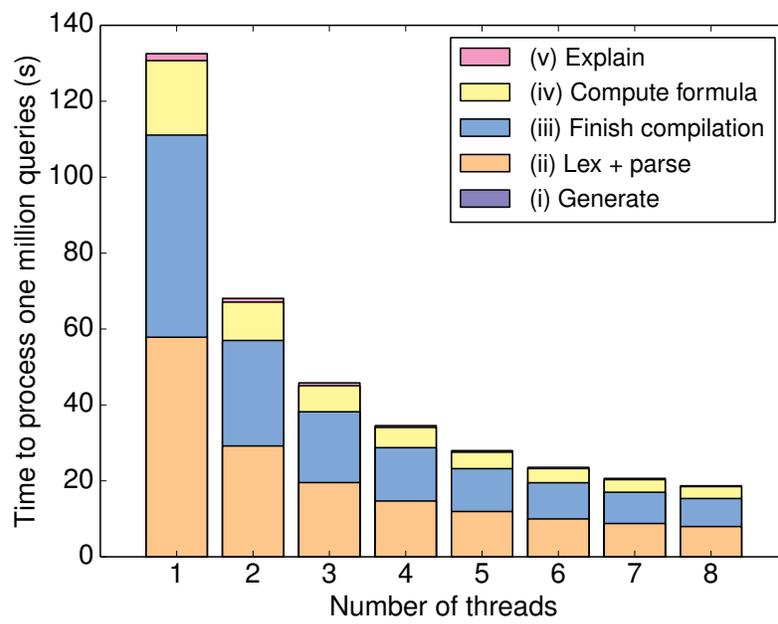


Figure 7.6: Performance for complex SQL features

CHAPTER 8

CONCLUSION

Although there has been a great deal of work on database security in the past, security constraints do not exist in a vacuum. They are typically defined and maintained by people, and are therefore prone to error. The problem is exacerbated on platforms such as Android and Facebook Apps: not only are the permission structures used by these systems complex, but app developers and end users are not required to undergo any kind of formal security training.

Although the need for reliable security mechanisms is broadly understood, it is insufficient on its own: we must also reduce opportunities for human error, both for developers and for end users. *Explainable security* represents an attempt to reduce developer error by providing app developers with *explanations* that indicate what permissions their apps should be requesting based on the queries they execute.

Explainable security is built on the theory of disclosure labeling, which we formally introduced in Chapter 3 and extended in Chapters 4 and 5. Disclosure labeling allows us to measure the information needed to answer a query Q in terms of a set of security views \mathbf{V} . Disclosure labeling allows us to characterize precisely which subsets of the views in \mathbf{V} would be sufficient to answer Q , and this serves as the basis for *explanations* that can guide principals towards requesting the right permissions for their queries.

Although the experiments in Section 7.2 suggest that explainable security is viable, a significant amount of work remains before explanation generation mechanisms can be integrated into existing systems. One key challenge is that many access control mechanisms that are used in practice, including those used by Facebook Apps, rely on Truman rather than non-Truman view rewriting. (See Section 2.3.) Rather than rejecting

queries outright, such mechanisms will semantically modify arbitrary SQL queries into queries that are known to be comply with a security policy. Since queries are *always* answered in such systems, it is unclear how our *explanations* should be adapted to such models. We leave this as future work.

BIBLIOGRAPHY

- [1] <https://github.com/JSQParser/JSqParser>.
- [2] Android permissions. <http://developer.android.com/guide/topics/security/permissions.html>.
- [3] Box OneCloud. <http://developers.box.com/onecloud/>.
- [4] Facebook Query Language. <https://developers.facebook.com/docs/reference/fql/>.
- [5] Grant – oracle database sql reference 10g release 2 (10.2). http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_9013.htm.
- [6] Grant – postgresql 9.1.13 documentation. <http://www.postgresql.org/docs/9.1/static/sql-grant.html>.
- [7] Grant syntax. <http://dev.mysql.com/doc/refman/5.1/en/grant.html>.
- [8] Grant (transact-sql). <http://msdn.microsoft.com/en-us/library/ms187965.aspx>.
- [9] Label-based access control (lbac) overview. <http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=%2Fcom.ibm.db2.udb.admin.doc%2Fdoc%2Fc0021114.htm>.
- [10] Overwarch SoldierEyes. <http://www.overwatch.com/products/soldiereyes.php>.
- [11] Virtual private database. <http://www.oracle.com/technetwork/database/security/index-088277.html>.
- [12] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [13] Moritz Y Becker, Cédric Fournet, and Andrew D Gordon. Secpal: Design and semantics of a decentralized authorization language. *CSF*, 2006.
- [14] Gabriel Bender, Łucja Kot, and Johannes Gehrke. Explainable security for relational databases. *SIGMOD*, 2014. <http://doi.acm.org/10.1145/2588555.2593663>.

- [15] Gabriel M Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. *SIGMOD*, 2013. <http://doi.acm.org/10.1145/2463676.2467798>.
- [16] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *TISSEC*, 2002.
- [17] D.F.C. Brewer and M.J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, 1989.
- [18] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *TKDE*, 12(6):900–919, 2000.
- [19] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. *STOC*, 1977.
- [20] Surajit Chaudhuri, Tanmoy Dutta, and S Sudarshan. Fine grained authorization through predicated grants. *ICDE*, 2007.
- [21] Surajit Chaudhuri and Moshe Y Vardi. Optimization of real conjunctive queries. *SIGMOD*, 1993.
- [22] Michael R Clarkson and Fred B Schneider. Quantification of integrity. In *CSF*, 2010.
- [23] Sara Cohen. Equivalence of queries that are sensitive to multiplicities. *VLDB*, 2009.
- [24] Michael Compton. Finding equivalent rewritings with exact views. *ICDE*, 2009.
- [25] B.A. Davey and H.A. Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.
- [26] Cynthia Dwork. Differential privacy. In *Automata, languages and programming*, pages 1–12. Springer, 2006.
- [27] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. *NDSS*, 2011.
- [28] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. *CCS*, 2011.

- [29] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. 2012.
- [30] A.P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *USENIX 2011*, 2011.
- [31] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD*, 2001.
- [32] Georg Gottlob and Christian G Fermüller. Removing redundancy from a clause. *Artificial Intelligence*, 1993.
- [33] Alon Y Halevy. Theory of answering queries using views. *SIGMOD*, 2000.
- [34] Daniel Kifer and Ashwin Machanavajjhala. No free lunch in data privacy. *SIGMOD*, 2011.
- [35] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Query-based data pricing. *PODS*, 2012.
- [36] Jaisook Landauer and Timothy Redmond. A lattice of information. *CSFW*, 1993.
- [37] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovic, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. Limiting disclosure in hippocratic databases. *VLDB*, 2004.
- [38] Alon Y Levy, Alberto O Mendelzon, and Yehoshua Sagiv. Answering queries using views. *SIGMOD*, 1995.
- [39] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkatasubramanian. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):3, 2007.
- [40] Gerome Miklau and Dan Suciu. A formal analysis of information disclosure in data exchange. *J. Comput. Syst. Sci.*, 73(3), 2007.
- [41] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *TODS*, 2010.
- [42] Lars E Olson, Carl A Gunter, and P Madhusudan. A formal framework for reflective database access control policies. *CCS*, 2008.

- [43] Nicole Perlroth. LinkedIn’s leaky mobile app has access to your meeting notes. *New York Times*, June 5 2012.
- [44] Rachel Pottinger and Alon Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB*, 10, 2001.
- [45] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3rd edition, 2003.
- [46] Shariq Rizvi, Alberto Mendelzon, Sundararajao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. *SIGMOD*, 2004.
- [47] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
- [48] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [49] Fred B Schneider. Language-based security for malicious mobile code. Technical report, DTIC Document, 2007.
- [50] Fred B Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (nal): Design rationale and applications. *TISSEC*, 14(1), 2011.
- [51] David Andrew Schultz. *Decentralized Information Flow Control for Databases*. Dissertation, MIT, 2012.
- [52] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.
- [53] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. *VLDB*, 2007.