2 Manipulating Query Expressions

The first step in answering a query posed to a data integration system is selecting which data sources are most relevant to the query. Making this decision relies on a set of methods for manipulating queries and reasoning about relationships between queries. This chapter covers such reasoning techniques in detail. While these techniques are very useful in data integration, they are also of independent interest and have been used in other contexts as well, such as query optimization and physical database design.

Section 2.1 begins with a review of database concepts that are used throughout the book. Section 2.2 describes techniques for *query unfolding*, where the goal is to reformulate a query, possibly posed over *other* queries or views, so that it refers only to the database relations. In the context of query optimization, query unfolding may let a query optimizer discover more efficient query processing plans because there is more freedom deciding in which order to perform join operations. In the context of data integration, query unfolding will be used to reformulate a query posed over a mediated schema into a query referring to data sources (Section 3.2.2).

Section 2.3 describes algorithms for *query containment* and *query equivalence*. Query containment is a fundamental ordering relationship that may hold between a pair of queries. If the query Q_1 contains the query Q_2, then the answers to Q_1 will always be a superset of the answers of Q_2 *regardless* of the state of the database. If two queries are equivalent, then they always produce the same answers, even though they may look different syntactically. We will use query containment and equivalence to decide whether two data sources are redundant with each other and whether a data source can be used to answer a query. In the context of query optimization, we use query equivalence to verify that a transformation on a query preserves its meaning.

Section 2.4 describes techniques for *answering queries using views*. Intuitively, answering queries using views considers the following problem. Suppose you have stored the results of several queries into a set of views over a database. Now you receive a new query and you want to know whether you can answer the query using *only* the views, without accessing the original database. In the context of query optimization, finding a way to answer a query using a set of views can substantially reduce the amount of computation needed to answer the query. As we describe in Section 3.2.3, in the context of data integration we often describe a set of data sources as views over a mediated schema. User queries are formulated using the terms in the mediated schema. Hence, answering queries using views is necessary in order to reformulate user queries to refer to the data sources.

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00002-8

2.1 Review of Database Concepts

We begin with a review of a few basic terms and concepts from the database literature related to data modeling and querying.

2.1.1 Data Model

Data integration systems need to handle data in a variety of data models, be it relational, XML, or unstructured data. We review the basics of the relational data model here, and introduce other data models later in the book. In particular, Chapter 11 discusses XML and its underlying data model, as it plays a key role in many data integration scenarios.

A relational database is a set of *relations*, also called *tables* (see Figure 2.1). A *database schema* includes a *relational schema* for each of its tables and a set of *integrity constraints* that we describe a bit later.

A *relational schema* specifies the set of attributes in the table and a data type for each attribute. The *arity* of a relation is the number of attributes it has. For example, in Figure 2.1, the arity of the Interview relation is 5 and its schema is:

candidate: string　　date: date
recruiter: string　　hireDecision: boolean
grade: float

In a sense, the schema describes how the database author decided to organize the data, what aspects of the data she chose to model, and the distinctions she wished to make. For example, the Interview table does not include an attribute for the location of the interview

Interview

candidate	date	recruiter	hireDecision	grade
Alan Jones	5/4/2006	Annette Young	No	2.8
Amanda Lucky	8/8/2008	Bob Young	Yes	3.7

EmployeePerformance

empID	name	reviewQuarter	grade	reviewer
2335	Amanda Lucky	1/2007	3.5	Eric Brown
5443	Theodore Lanky	2/2007	3.2	Bob Jones

Employee

empID	name	hireDate	manager
2335	Amanda Lucky	9/13/2005	Karina Lillberg
5443	Theodore Lanky	11/26/2004	Kasper Lillholm

FIGURE 2.1 The Interview table stores the employment candidates and the result of their interviews. The EmployeePerformance table describes the quarterly evaluation score of each employee.

or the position for which the candidate was interviewing. The EmployeePerformance table only provides a single grade, and does not model the fact that this grade may be the composition of several more specific performance measures. One of the challenges we face in data integration is that different sources organize their data differently, and these differences need to be reconciled.

A relational table includes a finite number of rows, called *tuples* (or *records*). A tuple assigns a value to each attribute of the table. If the relation being discussed is clear from the context, we denote its tuples with its values in parentheses, e.g.,

(Alan Jones, 5/4/2006, Annette Young, No, 2.8)

If not, we denote it as a *ground atom*:

Interview(AlanJones, 5/4/2006, AnnetteYoung, No, 2.8)

In some cases, we describe a tuple with a mapping from attribute names to values, such as:

{candidate→ Alan Jones, date → 5/4/2006, recruiter→ Annette Young,
hireDecision→ No, grade → 2.8}

We pay special attention to the NULL value in a database. Intuitively, NULL means that the value is not known or may not exist. For example, the value of an age attribute may be NULL if it's not known, whereas the value of spouse could be either unknown or may not exist. The important property to keep in mind about the NULL value is that an equality test involving a NULL returns NULL. In fact, even $NULL = NULL$ returns NULL. This makes intuitive sense because if two values are not known, we certainly do not know that they are equal to each other. We can test explicitly for NULL with the predicate `is NULL`.

A *state* of the database, or *database instance*, is a particular snapshot of the contents of the database. We distinguish between *set* semantics of databases and *multi-set* semantics. In set semantics, a database state assigns a set of tuples to each relation in the database. That is, a tuple can only appear once in a relation. In multi-set semantics, a tuple can appear any number of times in each relation, and hence a database instance is an assignment of a multi-set of tuples to each relation. Unless we state otherwise, our discussion will assume set semantics. Commercial relational databases support both semantics.

We use these notations throughout the book.

- We denote a database instance with D (possibly with subscripts).
- We denote attributes with letters from the beginning of the alphabet, e.g., A, B, C. We denote sets or lists of attributes with overbars, e.g., $\bar{A}$.
- We denote relation names with the letters R, S, and T and sets or lists of relations with overbars, e.g., $\bar{R}$.
- We denote tuples with the lowercase letters, e.g., s, t.
- If $\bar{A}$ is a set of attributes and t is a tuple in a relation that has the attributes in $\bar{A}$, then $t^{\bar{A}}$ denotes the restriction of the tuple t to the attributes in $\bar{A}$.

2.1.2 Integrity Constraints

Integrity constraints are a mechanism for limiting the possible states of the database. For example, in the employee database, we do not want two rows for the same employee. An integrity constraint would specify that in the employee table the employee ID needs to be unique across the rows. The languages for specifying integrity constraints can get quite involved. Here we focus on the following most common types of integrity constraints:

- **Key constraints:** A set of attributes $\bar{A}$ of a relation R is said to be a key of R if there do not exist a pair of tuples $t_1, t_2 \in R$ such that $t_1^{\bar{A}} = t_2^{\bar{A}}$ and $t_1 \neq t_2$. For example, the attribute candidate can be a key of the Interview table.
- **Functional dependencies:** We say that a set of attributes $\bar{A}$ functionally determines a set of attributes $\bar{B}$ in a relation R if for every pair of tuples $t_1, t_2 \in R$, if $t_1^{\bar{A}} = t_2^{\bar{A}}$, then $t_1^{\bar{B}} = t_2^{\bar{B}}$.

  For example, in the EmployeePerformance table, empID and reviewQuarter functionally determine grade. Note that a key constraint is implies a functional dependency where the key attributes determine all of the other attributes in the relation.
- **Foreign key constraints:** Let S be a relation with attribute A, and let T be a relation whose key is the attribute B. The attribute A is said to be a foreign key of attribute B of table T if whenever there is a row in S where the attribute A has the value v, then there must exist a row in T where the value of the attribute B is v. For example, the empID attribute of the EmployeePerformance table would be a foreign key of the Employee table.

GENERAL CONSTRAINT EXPRESSIONS

Tuple-generating dependencies (TGDs) and equality-generating dependencies (EGDs) offer a general formalism for specifying a wide class of integrity constraints. As we see in subsequent chapters, TGDs are also used to specify schema mappings.

Tuple-generating dependencies are formulas of the form

$$(\forall \bar{X}) s_1(\bar{X}_1), \ldots, s_m(\bar{X}_m) \rightarrow (\exists \bar{Y}) t_1(\bar{Y}_1), \ldots, t_l(\bar{Y}_l)$$

where $s_1, \ldots, s_m$ and $t_1, \ldots, t_l$ are relation names. The variables $\bar{X}$ are a subset of $\bar{X}_1 \ldots \cup \ldots \bar{X}_m$, and the variables $\bar{Y}$ are a subset of $\bar{Y}_1 \ldots \cup \ldots \bar{Y}_l$. The variables $\bar{Y}$ do not appear in $\bar{X}_1 \ldots \cup \ldots \bar{X}_m$. Depending on the context, the relation names on the left- and right-hand side of the dependency may refer to relations in the same schema or may refer to different databases.

Equality-generating dependencies are similar, except that the right-hand side contains only equality atoms:

$$(\forall \bar{Y}) t_1(\bar{Y}_1), \ldots, t_l(\bar{Y}_l) \rightarrow (Y_i^1 = Y_j^1), \ldots, (Y_i^k = Y_j^K)$$

where all the variables on the right-hand side appear on the left-hand side. We note that in practice, the quantifiers ($\forall \bar{X}$ and $\exists \bar{Y}$) are often omitted. In these cases, all the variables that appear on the left-hand side are assumed to be universally quantified (∀) and the variables appearing only on the right-hand side are assumed to be existentially quantified (∃).

As we illustrate below, we can use these formalisms to express the contraint classes mentioned above.

The attribute candidate is a key of the Interview table: (the following formula specifies a key constraint assuming set semantics for the table)

$$(\forall C_1, D_1, R_1, H_1, G_1, D_2, R_2, H_2, G_2)$$
$$\text{Interview}(C_1, D_1, R_1, H_1, G_1), \text{Interview}(C_1, D_2, R_2, H_2, G_2) \rightarrow$$
$$D_1 = D_2, R_1 = R_2, H_1 = H_2, G_1 = G_2$$

The attributes empID and reviewQuarter functionally determine grade in the EmployeePerformance table:

$$(\forall I, N_1, R, G_1, Re_1, N_2, G_2, Re_2)$$
$$\text{EmployeePerformance}(I, N_1, R, G_1, Re_1), \text{EmployeePerformance}(I, N_2, R, G_2, Re_2)$$
$$\rightarrow G_1 = G_2$$

The recruiter attribute of the Interview table is a foreign key of the EmployeePerformance table:

$$(\forall I, N, R, Gr, Re)\ \text{EmployeePerformance}(I, N, R, Gr, Re) \rightarrow (\exists Na, Hd, Mg)\ \text{Employee}(I, Na, Hd, Ma)$$

2.1.3 Queries and Answers

Queries are used for several purposes in data integration systems. As in database systems, queries are used in order to formulate users' information needs. In some cases, we may want to reuse the query expression in other queries, in which case we define a named *view* over the database, defined by the query. If we want the database system to compute the answer to the view and maintain the answer as the database changes, we refer to the view as a *materialized view*.

In data integration systems, we also use queries to specify *relationships* between the schemata of data sources. In fact, as we discuss in Chapter 3, queries form the core of the formalisms for specifying semantic mappings.

We distinguish between *structured* queries and *unstructured* queries. Structured queries such as SQL queries over relational databases or XQuery queries over XML databases are the ones we work hard to support in database systems. Unstructured queries are the ones we are most familiar with on the Web: the most common form of an unstructured query is a list of keywords.

We use two different notations for queries over relational databases throughout the book. The first is SQL, which is the language used to query relational data in commercial relational systems. Unfortunately, SQL is not known for its aesthetic aspects and hence not convenient for more formal expositions. Hence, in some of the more formal discussions, we use the notation of *conjunctive queries*, which is based on (a very simple form of) mathematical logic.

SQL is a very complex language. For our discussion, we typically use only its most basic features: selecting specific rows from a table, selecting specific columns from a table, combining data from multiple tables using the join operator, taking the union of two tables, and computing basic aggregation functions. For example, the following queries are typical of the ones we see in the book.

Example 2.1

```
SELECT recruiter, candidate
FROM Interview, EmployeePerformance
WHERE recruiter = name AND EmployeePerformance.grade < 2.5
```

Example 2.2

```
SELECT reviewer, AVG(grade)
FROM EmployeePerformance
WHERE reviewQuarter = "1/2007"
```

The query in Example 2.1 asks for pairs of recruiters and candidates where the recruiter got a low grade on their performance review. The answer to this query may reveal candidates who we may want to reinterview. The query in Example 2.2 asks for the average grade that a reviewer gave in the first quarter of 2007.

Given a query Q and a database D, we denote by $Q(D)$ the result of applying the query Q to the database D. Recall that $Q(D)$ is also a relation whose schema is defined implicitly by the query expression.

2.1.4 Conjunctive Queries

We briefly review the formalism for conjunctive queries. A conjunctive query has the following form:

$$Q(\bar{X})\text{:-}R_1(\bar{X}_1),\ldots,R_n(\bar{X}_n),c_1,\ldots,c_m$$

In the query, $R_1(\bar{X}_1),\ldots,R_n(\bar{X}_n)$ are the *subgoals* (or *conjuncts*) of the query and together form the *body* of the query. The R_i's are database relations, and the $\bar{X}$'s are tuples of variables and constants. Note that the same database relation can occur in multiple subgoals. Unless we explicitly give the query a different name, we refer to it as Q.

The variables in $\bar{X}$ are called *distinguished variables*, or *head variables*, and the others are *existential variables*. The predicate Q denotes the answer relation of the query. Its arity is the number of elements in $\bar{X}$. We denote by $Vars(Q)$ the set of variables that appear in its head or body.

The c_j's are *interpreted atoms* and are of the form $X\ \theta\ Y$, where X and Y are either variables or constants, and at least one of X or Y is a variable. The operator θ is an interpreted predicate such as $=, \leq, <, !=, >$, or $\geq$. We assume the obvious meaning for the interpreted predicates, and unless otherwise stated, we interpret them over a dense domain.[1]

The semantics of a conjunctive query Q over a database instance D is as follows. Consider any mapping ψ that maps each of the variables in Q to constants in D. Denote by $\psi(R_i)$ the result of applying ψ to $R_i(\bar{X}_i)$, by $\psi(c_i)$ the result of applying ψ to c_i, and by $\psi(Q)$ the result of applying ψ to $Q(\bar{X})$, all resulting in ground atoms. If

- each of $\psi(R_1), \ldots, \psi(R_n)$ is in D, and
- for each $1 \leq j \leq m$, $\psi(c_j)$ is satisfied,

then (and only then) $\psi(Q)$ is in the answer to Q over D.

To illustrate the correspondence between SQL queries and conjunctive queries, the following conjunctive query is the same as the SQL query in Example 2.1:

$$Q_1(\mathsf{Y}, \mathsf{X}) \text{ :- } \mathsf{Interview}(\mathsf{X}, \mathsf{D}, \mathsf{Y}, \mathsf{H}, \mathsf{F}), \mathsf{EmployeePerformance}(\mathsf{E}, \mathsf{Y}, \mathsf{T}, \mathsf{W}, \mathsf{Z}), \mathrm{W} \leq 2.5$$

Note that the join in the conjunctive query is expressed by the fact that the variable Y appears in both subgoals. The predicate on the grade is expressed with an interpreted atom.

Conjunctive queries must be *safe*; that is, every variable appearing in the head also appears in a non-interpreted atom in the body. Otherwise, the set of possible answers to the query may be infinite (i.e., the variable appearing in the head but not in the body can be bound to any value).

We can also express disjunctive queries in this notation. To express disjunction, we write two (or more) conjunctive queries with the same head predicate.

Example 2.3

The following query asks for the recruiters who performed the best *or* the worst:

$$Q_1(\mathsf{E}, \mathsf{Y}) \text{ :- } \mathsf{Interview}(\mathsf{X}, \mathsf{D}, \mathsf{Y}, \mathsf{H}, \mathsf{F}), \mathsf{EmployeePerformance}(\mathsf{E}, \mathsf{Y}, \mathsf{T}, \mathsf{W}, \mathsf{Z}), \mathrm{W} \leq 2.5$$
$$Q_1(\mathsf{E}, \mathsf{Y}) \text{ :- } \mathsf{Interview}(\mathsf{X}, \mathsf{D}, \mathsf{Y}, \mathsf{H}, \mathsf{F}), \mathsf{EmployeePerformance}(\mathsf{E}, \mathsf{Y}, \mathsf{T}, \mathsf{W}, \mathsf{Z}), \mathrm{W} \geq 3.9$$

We also consider conjunctive queries with negated subgoals, of the form

$$Q(\bar{X})\text{:-}R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), \neg S_1(\bar{Y}_1), \ldots, \neg S_m(\bar{Y}_m)$$

For queries with negation, we extend the notion of safety as follows: any variable appearing in the head of the query must also appear in a *positive* subgoal. To produce an answer for the query, the mapping from the variables of Q to the constants in the database must satisfy $\psi(S_1(\bar{Y}_1)), \ldots, \psi(S_m(\bar{Y}_m)) \notin D$.

[1]The alternative would be to interpret them over a discrete domain such as the integers. In that case we need to account for subtle inferences such as implying $X = 4$ from the conjunction $X > 3, X < 5$.

In our discussions, the term *conjunctive queries* will refer to conjunctive queries *without* interpreted predicates or negation. If we allow interpreted or negated atoms, we will explicitly say so.

2.1.5 Datalog Programs

A datalog program is a set of rules, each of which is a conjunctive query. Instead of computing a single answer relation, a datalog program computes a set of *intensional* relations (called IDB relations), one of them being designated as the *query predicate*. In datalog, we refer to the database relations as the *EDB relations* (extensional database). Intuitively, the extensional relations are given as a set of tuples (also referred to as ground facts), while the intensional relations are defined by a set of rules. Consequently, the EDB relations can occur in only the body of the rules, whereas the IDB relations can occur both in the head and in the body.

Example 2.4

Consider a database that includes a simple binary relation representing the edges in a graph: edge(X,Y) holds if there is an edge from X to Y in the graph. The following datalog query computes the paths in the graph. edge is an EDB relation and path is an IDB relation.

r_1 path(X, Y) :- edge(X, Y)
r_2 path(X, Y) :- edge(X, Z), path(Z, Y)

The first rule states that all single edges form paths. The second rule computes paths that are composed from shorter ones. The query predicate in this example is path. Note that replacing r_2 with the following rule would produce the same result.

r_3 path(X, Y) :- path(X, Z), path(Z, Y)

The semantics of datalog programs are based on conjunctive queries. We begin with empty extensions for the IDB predicates. We choose a rule in the program and apply it to the current extension of the EDB and IDB relations. We add the tuples computed for the head of the rule to its extension. We continue applying the rules of the program until no new tuples are computed for the IDB relations. The answer to the query is the extension of the query predicate. When the rules do not contain negated subgoals, this process is guaranteed to terminate with a unique answer, independent of the order in which we applied the rules.

Example 2.5

In Example 2.4, suppose we begin with a database that contains the tuples edge(1,2), edge(2,3), and edge(3,4). When we apply r_1 we will obtain path(1,2), path(2,3), and path(3,4). The first time we apply r_2 we obtain path(1,3) and path(2,4). The second time we apply r_2 we obtain path(1,4). Since no new tuples can be derived, the evaluation of the datalog program terminates.

In data integration we are interested in datalog programs mostly because they are sometimes needed in order to compute all the answers to a query from a set of data sources (see Sections 3.3 and 3.4). Readers who are familiar with the Prolog programming language will notice that datalog is a subset of Prolog. The reader should also note that not all SQL queries can be expressed in datalog. In particular, there is no support in datalog for grouping and aggregation and for outer joins. SQL does support limited kinds of recursion but not arbitrary recursion.

2.2 Query Unfolding

One of the important advantages of declarative query languages is *composability*: you can write queries that refer to views (i.e., other named queries) in their body. For example, in SQL, you can refer to other views in the `FROM` clause. Composability considerably simplifies the task of expressing complex queries, because they can be written in smaller fragments and combined appropriately. Query unfolding is the process of undoing the composition of queries: given a query that refers to views, query unfolding will rewrite the query so it refers only to the database tables.

Query unfolding is conceptually quite simple. We iteratively unfold a single view in the definition of the query until no more views remain. The following describes a single unfolding step. Like all other algorithms in this chapter, we describe them using the notation of conjunctive queries (see Section 2.1). We typically restrict our discussion to algorithms for manipulating conjunctive queries, and in some cases describe some important extensions. The bibliographic references contain pointers to algorithms that cover more complex queries.

UNFOLDING A SUBGOAL

Let Q be a conjunctive query of the form

$$Q(\bar{X}) \text{ :- } p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$$

where p_1 is itself a relation defined by the query

$$p_1(\bar{Y}) \text{ :- } s_1(\bar{Y}_1), \ldots, s_m(\bar{Y}_m)$$

We assume without loss of generality that $\bar{Y}$ is a tuple of variables, and each variable occurs in the tuple at most once. The other subgoals of Q may also be defined by other queries or be database relations.

A single unfolding step is performed as follows. Let ψ be the variable mapping that maps $\bar{Y}$ to $\bar{X}_1$ and maps the existential variables in p_1 to new variables that do not occur anywhere else. To unfold $p_1(\bar{X}_1)$, remove $p_1(\bar{X}_1)$ from Q and add the subgoals $s_1(\psi(\bar{Y}_1)), \ldots,$ $s_m(\psi(\bar{Y}_m))$ to the body of Q.

We repeat the above procedure until all of the subgoals in Q refer to database relations.

Example 2.6

Consider the following example, where Q_3 is defined in terms of Q_1 and Q_2. The relation Flight stores pairs of cities between which there is a direct flight, and the relation Hub stores the set of hub cities of the airline. The query Q_1 asks for pairs of cities between which there is a flight that goes through a hub. The query Q_2 asks for pairs of cities that are on the same outgoing path from a hub.

Q_1(X, Y) :- Flight(X, Z), Hub(Z), Flight(Z, Y)

Q_2(X, Y) :- Hub(Z), Flight(Z, X), Flight(X, Y)

Q_3(X, Z) :- Q_1(X, Y), Q_2(Y, Z)

The unfolding of Q_3 is

Q'_3(X, Z) :- Flight(X, U), Hub(U), Flight(U, Y), Hub(W), Flight(W, Y), Flight(Y, Z)

There are a few points to note about query unfolding. First, the resulting query may have subgoals that seem redundant. The next section will describe a set of algorithms for removing redundant subgoals by leveraging techniques for query containment. Second, the query and the view may each have *interpreted predicates* (recall from Section 2.1.4) that are satisfiable in isolation, but after the unfolding we may discover that the query is unsatisfiable and therefore the empty answer can be returned immediately. This, of course, is an extreme example where unfolding can lead to significant optimization in query evaluation.

It is also interesting to note that through repeated applications of the unfolding step, the number of subgoals may grow exponentially. It is quite easy to create an example of n queries defining a query Q, such that unfolding Q yields 2^n subgoals.

Finally, we emphasize that unfolding does not necessarily yield more efficient ways to execute the query. In fact, in Section 2.4 we do exactly the opposite—we try to rewrite queries so they *do* refer to views in order to speed up query processing. Unfolding merely allows the query processor to explore a wider collection of query plans by considering a larger set of possible orderings of the join operations in the query, and by considering the complete set of constraints expressed with interpreted predicates holistically. Of course, the ability to unfold queries is crucial when queries are written in a compositional fashion, which is one of the main benefits of declarative querying.

2.3 Query Containment and Equivalence

Let us reconsider the unfolding of the query Q_3 in Example 2.6:

Q'_3(X, Z) :- Flight(X, U), Hub(U), Flight(U, Y), Hub(W), Flight(W, Y), Flight(Y, Z)

Intuitively, this query seems to have more subgoals than necessary. Specifically, the subgoals Hub(W) and Flight(W, Y) seem redundant, because whenever the subgoals Hub(Z) and Flight(Z, Y) are satisfied, then so must Hub(W) and Flight(W, Y). Hence, we would expect the following query to produce exactly the same result as Q'_3:

Q_4(X, Z) :- Flight(X, U), Hub(U), Flight(U, Y), Flight(Y, Z)

Furthermore, if we consider the following query, which requires *both* intermediate stops to be hubs:

Q_5(X, Z) :- Flight(X, U), Hub(U), Flight(U, Y), Hub(Y), Flight(Y, Z)

then we would expect the set of answers to Q_4 to always be a *superset* of the answers to Q_5.

Query containment and equivalence provide a formal framework for reaching the conclusions we described above. Reasoning in this way enables us to remove subgoals from queries, thereby reducing the computation needed to execute them. As we will see in the next chapter, query containment and equivalence provide the formal framework for comparing different results of query reformulation in data integration systems.

2.3.1 Formal Definition

We begin with the formal definition. Recall that $Q(D)$ denotes the result of the query Q over the database D. The arity of a query refers to the number of arguments in its head.

Definition 2.1 (Containment and Equivalence). *Let Q_1 and Q_2 be two queries of the same arity. We say that Q_1 is contained in Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for any database D, $Q_1(D) \subseteq Q_2(D)$. We say that Q_1 is equivalent to Q_2, denoted by $Q_1 \equiv Q_2$, if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.* □

The important aspect of the above definition is that containment and equivalence are properties of the queries, and not the current state of the database. The relationship needs to hold for *any* database state. In fact, determining query containment and equivalence can be viewed as a problem of logical deduction specialized to query expressions.

In our discussion, we do not consider the data types of individual columns of database relations. However, in practice we would only consider containment between pairs of queries that are union compatible, i.e., each of the columns of their head predicates are compatible. In what follows, we describe query containment (and hence equivalence) algorithms for common classes of queries. The bibliographic references point out additional cases that have been studied and indicate where to find the full details of the algorithms we describe.

2.3.2 Containment of Conjunctive Queries

We begin by discussing the simplest case of query containment: conjunctive queries with no interpreted predicates or negation. In our discussion we will often refer to *variable mappings*. A variable mapping ψ from query Q_1 to query Q_2 maps the variables of Q_1 to either variables or constants in Q_2. We also apply variable mappings to tuples of variables and to atoms. Hence, $\psi(X_1,\ldots,X_n)$ denotes $\psi(X_1),\ldots,\psi(X_n)$, and $\psi(p(X_1,\ldots,X_n))$ denotes $p(\psi(X_1),\ldots,\psi(X_n))$.

In the case of conjunctive queries with no interpreted predicates or negation, checking containment amounts to finding a *containment mapping*, defined next.

Definition 2.2 (Containment Mapping). *Let Q_1 and Q_2 be conjunctive queries. Let ψ be a variable mapping from Q_1 to Q_2. We say that ψ is a containment mapping from Q_1 to Q_2 if*

- *$\psi(\bar{X}) = \bar{Y}$, where $\bar{X}$ and $\bar{Y}$ are the head variables of Q_1 and Q_2, respectively, and*
- *for every subgoal $g(\bar{X}_i)$ in the body of Q_1, $\psi(g(\bar{X}_i))$ is a subgoal of Q_2.* □

The following theorem shows that the existence of a containment mapping is a necessary and sufficient condition for containment.

Theorem 2.1. *Let Q_1 and Q_2 be two conjunctive queries. Then, $Q_1 \sqsupseteq Q_2$ if and only if there is a containment mapping from Q_1 to Q_2.* □

Proof. Suppose Q_1 and Q_2 have the following form:

$Q_1(\bar{\mathsf{X}})$:- $g_1(\bar{\mathsf{X}}_1), \ldots, g_n(\bar{\mathsf{X}}_n)$
$Q_2(\bar{\mathsf{Y}})$:- $h_1(\bar{\mathsf{Y}}_1), \ldots, h_m(\bar{\mathsf{Y}}_m)$

For the *if* direction, suppose there is a containment mapping, ψ, from Q_1 to Q_2 and suppose that D is an arbitrary database instance. We need to show that if $\bar{t} \in Q_2(D)$, then $\bar{t} \in Q_1(D)$.

Suppose $\bar{t} \in Q_2(D)$; then there is a mapping, ϕ, from the variables of Q_2 to the constants in D such that

1. $\phi(\bar{Y}) = \bar{t}$, and
2. $h_i(\phi(\bar{Y}_i)) \in D$ for $1 \le i \le m$.

Now consider the composition mapping $\phi \circ \psi$ that maps the variables of Q_1 to constants in D (i.e., the mapping that first applies ψ and then applies ϕ to the result). Since ψ is a containment mapping, the following conditions, necessary to show that $\bar{t} \in Q_1(D)$, hold:

- $\phi \circ \psi(\bar{X}) = \bar{t}$ because $\psi(\bar{X}) = \bar{Y}$ and $\phi(\bar{Y}) = \bar{t}$.
- For every $1 \le i \le n$, $g_i(\psi(\bar{X}_i))$ is a subgoal of Q_2, and therefore $g_i(\phi \circ \psi(\bar{X}_i)) \in D$.

Therefore, $\bar{t} \in Q_1(D)$.

For the *only if* direction, assume that $Q_1 \sqsupseteq Q_2$, and we will show that there is a containment mapping from Q_1 to Q_2.

Let us consider a special database, D_C, that we call the *canonical database* of Q_2, and is constructed as follows. The constants in D_C are the variables or constants appearing in the body of Q_2. The tuples of D_C correspond to the subgoals of Q_2. That is, for every $1 \le i \le m$, the tuple $\bar{Y}_i$ is in the relation h_i.

Clearly, $\bar{Y} \in Q_2(D_C)$, simply by the definition of Q_2. Since $Q_1 \sqsupseteq Q_2$, $\bar{Y}$ must also be in $Q_1(D_C)$, and therefore there is a mapping ψ from the variables of Q_1 to the constants in Q_2 such that

- $\psi(\bar{X}) = \bar{Y}$, and
- for every $1 \le i \le n$, $g_i(\psi(\bar{X}_i)) \in D_C$.

It is easy to see that ψ is a containment mapping from Q_1 to Q_2. □

The important aspect of the above proof is the concept of a canonical database. We were able to show that if $Q_1(D_C) \supseteq Q_2(D_C)$, then containment holds on *any* database. Hence, we obtain Algorithm 1 for checking containment.

Algorithm 1. CQContainment: Query containment for conjunctive queries.

Input: conjunctive query Q_1; conjunctive query Q_2. **Output:** returns **true** if $Q_1 \sqsupseteq Q_2$.
Let Q_1 be of the form $Q_1(\bar{X})$:- $g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$
Let Q_2 be of the form $Q_2(\bar{Y})$:- $h_1(\bar{Y}_1), \ldots, h_m(\bar{Y}_m)$
// Freeze Q_2:
Create a database D_C where the constants are the variables and constants in Q_2
for every $1 \le i \le m$ **do**
  add the tuple $\bar{Y}_i$ to the relation h_i
end for
Evaluate Q_1 over D_C
return true if and only if $\bar{X} \in Q_1(D_C)$.

Example 2.7

Recall the queries we considered earlier:

Q'_3(X,Z) :- Flight(X,U), Hub(U), Flight(U,Y), Hub(W), Flight(W,Y), Flight(Y,Z)
Q_4(X$_1$,Z$_1$) :- Flight(X$_1$,U$_1$), Hub(U$_1$), Flight(U$_1$,Y$_1$), Flight(Y$_1$,Z$_1$)

The following containment mapping shows that $Q'_3 \sqsupseteq Q_4$:

$$\{X \to X_1, Z \to Z_1, W \to U_1, Y \to Y_1\}$$

As we will see soon, a single canonical database will not always suffice as we consider queries with interpreted predicates or negation, but we will be able to remedy the situation by considering multiple canonical databases.

COMPUTATIONAL COMPLEXITY

Deciding whether $Q_1 \sqsupseteq Q_2$ is NP-complete in the size of the two queries. In practice, however, this is not a concern for multiple reasons. First, we are measuring the complexity in terms of the size of the queries, not the size of the data, and queries tend to be relatively small (though not always!). In fact, the algorithm above evaluates a query over an extremely small database. Second, in many practical cases, there are polynomial-time algorithms for containment. For example, if none of the database relations appears more than twice in the body of one of the queries, then it can be shown that containment can be checked in time that is polynomial in the size of the queries.

2.3.3 Unions of Conjunctive Queries

Next, we consider containment and equivalence of unions of conjunctive queries. Recall that unions are expressed as multiple rules with the same relation in the head.

Example 2.8

The following query asks for the pairs of cities that (1) are either connected by one-stop flight, or (2) both have a flight into the same hub.

Q_1(X,Y) :- Flight(X,Z), Flight(Z,Y)
Q_1(X,Y) :- Flight(X,Z), Flight(Y,Z), Hub(Z)

Suppose we want to decide whether the following query is contained in Q_1:

Q_2(X,Y) :- Flight(X,Z), Flight(Z,Y), Hub(Z)

The following theorem shows a very important property: if Q_2 is contained in Q_1, then there must be a single conjunctive query in Q_1 that contains Q_2 on its own. In other words, two conjunctive queries in Q_1 cannot "gang up" on Q_2.

Theorem 2.2. *Let $Q_1 = Q_1^1 \cup \ldots \cup Q_1^n$ be a union of conjunctive queries and let Q_2 be a conjunctive query. Then $Q_1 \sqsupseteq Q_2$ if and only if there is an $1 \leq i \leq n$, such that $Q_1^i \sqsupseteq Q_2$.* □

Proof. The *if* direction is obvious. If one of the conjunctive queries in Q_1 contains Q_2, then clearly $Q_1 \sqsupseteq Q_2$.

For the *only if* direction, we again consider the canonical database created by Q_2, D_C. Suppose the head of Q_2 is $\bar{Y}$. Since $Q_1 \sqsupseteq Q_2$, then $\bar{Y}$ should be in $Q_1(D_C)$, and therefore there is some $1 \leq i \leq n$, such that $\bar{Y} \in Q_1^i(D_C)$. It is easy to see that $Q_1^i \sqsupseteq Q_2$. □

In Example 2.8, containment holds because Q_2 is contained in the first rule defining Q_1.

An important corollary of Theorem 2.2 is that the algorithm for checking containment of queries with union is a slight variation on the previous algorithm. Specifically, we create a canonical database from the body of Q_2. If $\bar{X}$ is in the answer to Q_1 over the canonical database, then $Q_1 \sqsupseteq Q_2$. In the case that Q_2 is a union of conjunctive queries, $Q_1 \sqsupseteq Q_2$ if and only if Q_1 contains each of the conjunctive queries in Q_2. Hence, the complexity results for conjunctive queries transfer over to queries with unions.

2.3.4 Conjunctive Queries with Interpreted Predicates

We now consider conjunctive queries with interpreted predicates, which have the form:

$$Q(\bar{\mathsf{X}}) \text{ :- } R_1(\bar{\mathsf{X}}_1), \ldots, R_n(\bar{\mathsf{X}}_n), c_1, \ldots, c_m$$

where c_j's are *interpreted atoms* (we often call them *comparisons*), and are of the form $X\theta Y$, where X and Y are either variables or constants (but at least one of them must be a variable). The operator θ is an interpreted predicate such as $=$, $\leq$, $<$, $\neq$, $>$, or $\geq$. We assume the obvious meaning for the interpreted predicates, and unless otherwise stated, we interpret them over a dense domain.

Conjunctive queries allow conjunctions of interpreted atoms. In some of our reasoning, we will also manipulate Boolean formulas that include disjunctions. We use the standard notation for logical entailment. Specifically, if C is a Boolean formula over interpreted atoms, and c is an interpreted atom, then $C \models c$ means that any variable substitution that satisfies C also satisfies c. For example, $\{X \leq Y, Y \leq 5\} \models X \leq 5$, but $\{X \leq Y, Y \leq 5\} \not\models Y \leq 4$. Checking whether $C \models c$ can be done in time quadratic in the sizes of C and c.

The following definition extends the notion of containment mappings to queries with interpreted predicates.

Definition 2.3 (Containment Mapping with Interpreted Predicates). *Let Q_1 and Q_2 be conjunctive queries with interpreted atoms. Let C_1 (resp. C_2) be the conjunction of interpreted predicates in Q_1 (resp. Q_2). Let ψ be a variable mapping from Q_1 to Q_2. We say that ψ is a containment mapping from Q_1 to Q_2 if*

- *$\psi(\bar{X}) = \bar{Y}$, where $\bar{X}$ and $\bar{Y}$ are the head variables of Q_1 and Q_2, respectively,*
- *for every non-interpreted subgoal $g(\bar{X}_i)$ in the body of Q_1, $g(\psi(\bar{X}_i))$ is a subgoal of Q_2, and*
- *$C_2 \models \psi(C_1)$.* □

It is easy to verify that one direction of Theorem 2.1 extends to queries with interpreted predicates. That is, if there is a containment mapping from Q_1 to Q_2, then $Q_1 \sqsupseteq Q_2$. However, as the following example shows, the converse is not true.

Example 2.9

Q_1(X, Y) :- R(X, Y), S(U, V), U $\leq$ V

Q_2(X, Y) :- R(X, Y), S(U, V), S(V, U)

It is easy to see that $Q_1 \sqsupseteq Q_2$ because in either alternative, $U \leq V$ or $U > V$, the S subgoal in Q_1 will be satisfied. However, there is no containment mapping with interpreted predicates from Q_1 to Q_2.

In fact, the reasoning that shows that $Q_1 \sqsupseteq Q_2$ in Example 2.9 is the key to developing a containment algorithm for conjunctive queries with interpreted predicates. In the example, suppose we rewrite Q_2 as the following union:

Q_2(X, Y) :- R(X, Y), S(U, V), S(V, U), U $\leq$ V
Q_2(X, Y) :- R(X, Y), S(U, V), S(V, U), U > V

Now it is fairly easy to verify using containment mappings that Q_1 contains each of the two conjunctive queries in Q_2. The following discussion will make this intuition precise.

We first introduce *complete orderings* that are refinements of a set of conjunctive comparison atoms. Specifically, a conjunction of comparison predicates, C, may still only specify partial knowledge about the orderings of the variables in C. For example, $\{X \geq 5, Y \leq 8\}$ does not entail $X \leq Y$, $X \geq Y$, or $X = Y$. The complete orderings originating from C are the sets of comparisons that are consistent with C, but also completely determine all the order relations between pairs of variables. Naturally, there are multiple complete orderings consistent with a given conjunction, C.

Definition 2.4 (Complete Ordering and Query Refinements). *Let Q be a conjunctive query whose variables are $\bar{X} = X_1, \ldots, X_n$ and mentioning the constants $\bar{A} = a_1, \ldots, a_m$. Let C be the conjunction of interpreted atoms in Q. A complete ordering C_T on the variables $\bar{X}$ is a satisfiable conjunction of interpreted predicates, such that $C_T \models C$ and for every pair d_1, d_2, where $d_1, d_2 \in \bar{X} \cup \bar{A}$, one of the following holds:*

- $C_T \models d_1 < d_2$,
- $C_T \models d_1 > d_2$,
- $C_T \models d_1 = d_2$.

Given a conjunctive query

$Q(\bar{X})$:- $R_1(\bar{X}_1)$, ..., $R_n(\bar{X}_n)$, C

let $c^1, \ldots, c^l$ be the complete orderings of the variables and constants in Q. Then

$$Q^1(\bar{X}) :- R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), c^1$$
$$\vdots$$
$$Q^l(\bar{X}) :- R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), c^l$$

are the complete query refinements of Q. Note that $Q \equiv Q^1 \cup \ldots \cup Q^l$. □

To check that Q_1 contains Q_2, we need to find a containment mapping from Q_1 to each of the complete query refinements of Q_2. The following theorem states this intuition formally.

Theorem 2.3. *Let* Q_1 *and* Q_2 *be two conjunctive queries with interpreted predicates. Let* $Q_2^1, \ldots, Q_2^l$ *be the complete query refinements of* Q_2. $Q_1 \sqsupseteq Q_2$ *if and only if for every* $1 \leq i \leq l$ *there is a containment mapping from* Q_1 *to* Q_2^i. □

Proof. The *if* direction is rather obvious since $Q \equiv Q^1 \cup \ldots \cup Q^l$ and because containment mappings are a sufficient condition for containment even with interpreted predicates.

For the *only if* direction, consider l canonical databases, $C_D^1, \ldots, C_D^l$, each constructed as before, except that the constants in C_D^i are rational numbers that satisfy the interpreted predicates in Q_2^i. Let t_i be an answer derived by Q_2^i from C_D^i. Since $Q_1 \sqsupseteq Q_2$, then $t_i \in Q_1(C_D^i)$. A similar argument to the proof of Theorem 2.1 shows that there is a containment mapping from Q_1 to Q_2^i. □

The practical problem with the algorithm implied by Theorem 2.3 is that the number of complete query refinements of Q_2 can be quite large. In fact, the number of refinements can be exponential in the size of Q_2. Recall that the number of refinements corresponds to all the different orderings on constants and variables appearing in the query. As a result, the computational complexity of checking containment for conjunctive queries with interpreted predicates is Σ_2^p-complete, which is the class of problems described by formulas of the form $\forall X \exists Y P(X, Y)$, where P is a condition that can be verified in polynomial time. Here, X ranges over the set of refinements, Y ranges over the set of variable mappings, and P verifies that the variable mapping is a containment mapping.

Fortunately, in practice we do not need to consider all possible refinements of Q_2. Algorithm 2 describes, **CQIPContainment**, a more efficient algorithm for query containment with interpreted predicates.

Applying algorithm **CQIPContainment** to our example, we would evaluate Q_1 on the database D_C that contains the tuples $\mathbf{R}(X, Y), \mathbf{S}(U, V), \mathbf{S}(V, U)$. $Q_1(D_C)$ would contain the following pairs: $((X, Y), U \leq V), ((X, Y), U \geq V)$. Hence, $C = U \leq V \vee U \geq V = True$. Since $C_2 = True$, we get $C_2 \models C$, and therefore $Q_1 \sqsupseteq Q_2$.

2.3.5 Conjunctive Queries with Negation

Next, we consider queries with negation that have the following form:

$$Q(\bar{X}) :- R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), \neg S_1(\bar{Y}_1), \ldots, \neg S_m(\bar{Y}_m)$$

Algorithm 2. CQIPContainment: Query containment for conjunctive queries with interpreted predicates.

Input: conjunctive query Q_1; conjunctive query Q_2. **Output:** returns **true** if $Q_1 \sqsupseteq Q_2$.
Let Q_1 be of the form $Q_1(\bar{X})$:- $g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$
Let Q_2 be of the form $Q_2(\bar{Y})$:- $h_1(\bar{Y}_1), \ldots, h_m(\bar{Y}_m)$
// Freeze Q_2:
Create a database D_C where the constants are the variables and constants in Q_2
for every $1 \le i \le m$ **do**
  add the tuple $\bar{Y}_i$ to the relation h_i
end for
Evaluate $Q'_1(\bar{X}) : -g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$ over D_C
Every tuple in $Q'_1(D_C)$ can be described as a pair $(t, \phi(C_1))$, where ϕ is the variable mapping from Q'_1 to D_C that satisfied all the subgoals of Q'_1
Let $C = c_1 \vee \ldots \vee c_k$, where $(\bar{Y}, c_1), \ldots, (\bar{Y}, c_k)$ are all the tuples in $Q'_1(D_C)$ with $\bar{Y}$ as their first component
return $Q_1 \sqsupseteq Q_2$ if and only if (1) there is at least one tuple of the form $(\bar{Y}, c) \in Q'_1(D_C)$ and (2) $C_2 \models C$

We require that the queries be *safe*, i.e., that any variable appearing in the head of the query also appear in a positive subgoal. For ease of exposition, we consider queries without interpreted predicates.

The natural extension of containment mappings to queries with negation ensures that positive subgoals are mapped to positive subgoals and negative subgoals are mapped to negative subgoals. The following theorem shows that containment mappings offer a sufficient condition for containment. We leave the proof to the reader, as it is an extension of the proof of Theorem 2.1.

Theorem 2.4. *Let Q_1 and Q_2 be safe conjunctive queries with negated subgoals. If there is a containment mapping ψ from Q_1 to Q_2 such that ψ maps the positive subgoals of Q_1 to positive subgoals of Q_2 and maps the negative subgoals of Q_1 to negative subgoals of Q_2, then $Q_1 \sqsupseteq Q_2$.* □

Here, too, containment mappings do not provide a necessary condition for containment. It is actually quite tricky to find an example where containment holds but there is no containment mapping, but the following example illustrates this case.

Example 2.10

Consider the queries Q_1 and Q_2. Note that they do not have head variables, and are therefore Boolean queries—they return true or false.

$Q_1()$:- $a(\mathsf{A}, \mathsf{B}), a(\mathsf{C}, \mathsf{D}), \neg\, a(\mathsf{B}, \mathsf{C})$
$Q_2()$:- $a(\mathsf{X}, \mathsf{Y}), a(\mathsf{Y}, \mathsf{Z}), \neg\, a(\mathsf{X}, \mathsf{Z})$

A little bit of thought will show that $Q_1 \sqsupseteq Q_2$; however, there is no containment mapping from Q_1 to Q_2.

Recall that in the previous cases, the key to proving containment was to consider a very special canonical database (or set of databases). If containment was satisfied on all canonical databases, then we were able to derive that containment holds on *all* databases. In the case of queries with negated subgoals, the key to proving containment is to show that we only need to consider databases of a certain size.

Theorem 2.5. *Let Q_1 and Q_2 be two conjunctive queries with safe negated subgoals and assume they mention disjoint sets of variables. Let B be the total number of variables and constants in Q_2. Then, $Q_1 \sqsupseteq Q_2$ if and only if $Q_1(D) \supseteq Q_2(D)$ on all databases D that have at most B constants.* □

Proof. Let Q_1 and Q_2 be of the following form (we omit the variables in the subgoals):

$$Q_1(\bar{X}) \text{ :- } p_1, \ldots, p_n, \neg r_1, \ldots, \neg r_l$$
$$Q_2(\bar{Y}) \text{ :- } q_1, \ldots, p_m, \neg s_1, \ldots, \neg s_k$$

The *only if* direction is obvious, so we consider the *if* direction. Suppose $Q_1 \not\sqsupseteq Q_2$. It suffices to show that there is a counterexample database with at most B constants.

Since $Q_1 \not\sqsupseteq Q_2$, there must exist a database D, a tuple $\bar{t}$, and a mapping ϕ from the variables of Q_2 to the constants in D, such that (1) $\phi(\bar{Y}) = \bar{t}$, (2) $\phi(q_i) \in D$, for $1 \leq i \leq m$, (3) $\phi(s_1), \ldots, \phi(s_k) \notin D$ for $1 \leq i \leq k$, and (4) $\bar{t} \notin Q_1(D)$.

Consider the database D' that includes only the tuples from D that either have only constants in the range of ϕ or have constants from Q_2.

First, we argue that $\bar{t} \in Q_2(D')$. This follows from the construction of D'. The range of ϕ is unaffected, and therefore the positive atoms that contributed to deriving $\bar{t}$ are in D' and the negative atoms certainly hold because D' is a subset of D. Hence, (2) and (3) above still hold. In fact, the same argument shows that $\bar{t} \in Q_2(D'')$ for any database D'' such that $D' \subseteq D'' \subseteq D$.

Second, we argue that we can build a database D'', where $D' \subseteq D'' \subseteq D$ and D'' is a counterexample. Furthermore, the number of constants in D'' is at most B. Note that the number of constants in D' is bounded by B.

Let us consider two cases. In the first case, there was no mapping, ψ, from the variables of Q_1 to D, such that $\psi(p_i) \in D$ for $1 \leq i \leq n$. In this case, there certainly is no such mapping from Q_1 to D', and therefore D' is a counterexample because $\bar{t} \notin Q_1(D')$.

In the second case, there may be mappings from the variables of Q_1 to D' that satisfy its positive subgoals. We construct D'' as following, after initializing it to D'.

Let ψ be a mapping from the variables of Q_1 to D', such that $\psi(p_i) \in D'$ for $1 \leq i \leq n$. There must be a j, $1 \leq j \leq l$, such that $\psi(r_j) \in D$. Otherwise, $\bar{t}$ would be in the answer to $Q_1(D)$. We add $\psi(r_j)$ to D'', and therefore ψ is no longer a variable mapping that would lead to $\bar{t}$ being in $Q_1(D'')$. Note that adding $\psi(r_j)$ to D'' did not change the number of constants in D'' because all the variables that occur in r_j must occur in one of the p_i's.

Since the number of tuples we may add to D'' in this way is bounded, we will ultimately not be able to add any more tuples. The database D'' has at most B constants, and $\bar{t} \notin Q_1(D'')$. Hence, D'' is a counterexample, establishing the theorem. □

Theorem 2.5 entails that it suffices to check that containment holds on all databases with up to B constants (up to isomorphism). Since there are a finite number of such databases, we can conclude that query containment is decidable. In what follows we describe a strategy that is more efficient than naively enumerating all possible such databases.

Let C denote the variables appearing in Q_2. We construct a set of canonical databases whose constants are C. Intuitively, each canonical database corresponds to a set of equality constraints applied to the elements of C (in the same spirit of refinements in Section 2.3.4, but considering only the $=$ predicate).

Formally, in each canonical database we apply a partition to elements of C. A partition of C maps the constants in C to a set of equivalence classes and applies a homomorphism to C where each constant is mapped to a unique representative of its equivalence class. Let k be the number of possible partitions of C, and ϕ_k the homomorphism associated with the kth partition. We create databases $D_1, \ldots, D_k$, where the database D_i includes the tuples $\phi_k(q_1), \ldots, \phi_k(q_m)$. For each D_i we construct a set of databases $\mathcal{D}_i$ as follows:

- If the body of Q_2 is not satisfied by D_i, we set $\mathcal{D}_i$ to be the empty set. Note that D_i may not satisfy the body of Q_2 because a negative subgoal of Q_2 may not be satisfied.
- If the body of Q_2 is satisfied by D_i, then $\mathcal{D}_i$ is the set of canonical databases that can be constructed from D_i as follows:
  - Add any subset of tuples to D_i that includes only constants from D_i, but do not add the $\phi_i(s_1), \ldots, \phi_i(s_k)$.

The containment $Q_1 \sqsupseteq Q_2$ holds if and only if for every i, $1 \leq i \leq k$ and for every $D \in \mathcal{D}_i$, $\phi_i(\bar{X}) \in Q_1(D)$.

Example 2.11

We illustrate the containment algorithm on the queries in Example 2.10. Q_2 includes four constants, A, B, C, and D.

We first consider the partition in which all variables in Q_2 are equated, yielding the database $D_1 = \{a(\mathsf{A},\mathsf{A})\}$. The body of Q_2 is not satisfied on D_1 because the negative subgoal is not satisified. Hence, $\mathcal{D}_1$ is the empty set and containment holds trivially.

Now consider the other extreme, the partition in which each variable is in its own equivalence class, yielding the database $D_2 = \{a(\mathsf{A},\mathsf{B}), a(\mathsf{C},\mathsf{D})\}$. Since Q_2 is satisfied in D_2, $\mathcal{D}_2$ includes any database that can be constructed by adding tuples to D_2 that have the constants A, B, C, and D, but *not* the atom $a(\mathsf{B},\mathsf{C})$. A case-by-case analysis verifies that Q_1 is satisfied in every database in $\mathcal{D}_2$.

In the same fashion, the reader can verify containment holds when we consider the other partitions of the variables in Q_2.

COMPUTATIONAL COMPLEXITY

The computational complexity of checking containment for conjunctive queries with negated subgoals is Σ_2^p-complete, which is the class of problems described by formulas of the form $\forall X \exists Y P(X,Y)$, where P is a condition that can be verified in polynomial time. Here, X ranges over the set of canonical databases $\mathcal{D}_1,\ldots,\mathcal{D}_k$, Y ranges over the set of variable mappings, and P is the function that verifies that the variable mapping is a containment mapping.

2.3.6 Bag Semantics, Grouping, and Aggregation

SQL queries operate, by default, on bags (multi-sets) of tuples, rather than sets of tuples. Answers to queries in SQL are also bags by default, unless the `DISTINCT` keyword is used. The key difference in bag semantics is that we also count the number of times a tuple appears in a relation (either the input or the result of a query). We refer to that number as the tuple's *multiplicity*. For example, consider the following query over the table in Figure 2.2:

Q_1(X,Y) :- Flight(X,Z,W), Flight(Z,Y,W$_1$)

Under set semantics, the answer to Q_1 would contain a single tuple (San Francisco, Anchorage). Under bag semantics, the tuple (San Francisco, Anchorage) appears in the answer with multiplicity 2, because there are two pairs of flights that connect San Francisco with Anchorage.

The topic of query containment and equivalence for queries with bag semantics is quite involved. Query containment and equivalence for queries with grouping and aggregation depend heavily on our understanding of containment for bags. This section gives an overview of the issues and some of the results known in this area.

To discuss containment of queries with aggregation, we first need to modify the definition of containment slightly to account for the different semantics. Specifically, we say that $Q_1 \sqsupseteq Q_2$ with bag semantics if for any database D and for any tuple $\bar{t}$, the multiplicity of $\bar{t}$ in

Flight

Origin	Destination	DepartureTime
San Francisco	Seattle	8AM
San Francisco	Seattle	10AM
Seattle	Anchorage	1PM

FIGURE 2.2 A simple flight table that illustrates the difference between set and bag semantics.

$Q_1(D)$ is at least as much as the multiplicity of $\bar{t}$ in $Q_2(D)$. As before, $Q_1 \equiv Q_2$ are equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.

Example 2.12

As a trivial example showing that equivalence under set semantics does not imply equivalence under bag semantics, consider the following two queries:

Q_1(X) :- P(X)
Q_2(X) :- P(X), P(X)

Suppose an instance of P has n occurrences of a constant d. Then the result of Q_2 would contain n^2 occurrences of d, while Q_1 would only contain n occurrences of d. Hence, while Q_1 and Q_2 are equivalent under set semantics, they are not equivalent under bag semantics.

Query containment for conjunctive queries without interpreted predicates and without negation is not even known to be decidable. Furthermore, it is known that when interpreted predicates are allowed or when unions are allowed, query containment is undecidable. There is more known on query equivalence. In fact, the following theorem states that for two conjunctive queries to be bag-equivalent, they need to be isomorphic to each other. The bibliographic notes include a reference to the proofs of the theorems in this section.

Theorem 2.6. *Let Q_1 and Q_2 be conjunctive queries. Then, Q_1 and Q_2 are bag-equivalent if and only if there is a 1-1 containment mapping (i.e., isomorphism) from Q_1 to Q_2.* □

QUERIES WITH GROUPING AND AGGREGATION

Conditions for equivalence for queries with grouping and aggregation highly depend on the specific aggregation function in the query. Some aggregation functions are sensitive to multiplicities (e.g., Average and Count), while others are not (e.g., Max and Min). The following are two examples of where precise conditions are known for equivalence of such queries.

COUNT QUERIES

These queries are of the form

$Q(\bar{X}, \text{count}(\bar{Y}))$:- $R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n)$

The query computes the table specified by the body subgoals, groups the results by the variables in $\bar{X}$, and for every group outputs the number of different values for $\bar{Y}$.

The following theorem shows that count queries are sensitive to multiplicities. The theorem can be extended to queries with interpreted predicates using the concept of complete orderings we introduced earlier.

Theorem 2.7. *Let Q_1 and Q_2 be two count queries. The equivalence $Q_1 \equiv Q_2$ holds if and only if there is a variable mapping ψ from the variables of Q_1 to the variables of Q_2 that* *induces an isomorphism between the bodies of Q_1 and Q_2 and an isomorphism between the heads of Q_1 and Q_2.* □

MAX QUERIES

It can be shown that max queries are *not* sensitive to multiplicities. A max query has the form

$$Q(\bar{X}, \max(Y)) \text{ :- } R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n)$$

The query computes the table specified by the body subgoals, groups the results by the variables in $\bar{X}$, and for every group outputs the maximum value of the variable Y. We define the *core* of a max query to be the query where the variable Y appears in the head without the aggregation function, i.e.,

$$Q(\bar{X}, Y) \text{ :- } R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n)$$

The following theorem establishes the property of max queries:

Theorem 2.8. *Let Q_1 and Q_2 be two max queries with no comparison predicates. The equivalence $Q_1 \equiv Q_2$ holds if and only if the core of Q_1 is equivalent to the core of Q_2.* □

Example 2.13

The following example shows that Theorem 2.8 does not apply to queries with interpreted predicates.

$$Q_1(\max(Y)) \text{ :- } p(Y), P(Z_1), p(Z_2), Z_1 < Z_2$$
$$Q_2(\max(Y)) \text{ :- } p(Y), P(Z), Z < Y$$

Both queries return answers if P contains at least two different constants. However, while Q_1 considers all tuples in P, Q_2 considers all the tuples except for the one with the smallest value. Hence, the maximum value is always the same (and hence, $Q_1 \equiv Q_2$), but the cores are not equivalent. Fortunately, there is a more elaborate condition for checking containment of max queries with interpreted predicates (see the bibliographic references for more details).

2.4 Answering Queries Using Views

In the previous section we described query containment, which is a fundamental relationship between a pair of queries. One of the important uses of query containment is that when we detect that a query Q_1 is contained in a query Q_2, we can often compute the answer to Q_1 from the answer to Q_2. In this section we are interested in the more general problem of when we can answer a query Q from a collection of views. The following example illustrates our goal.

Example 2.14

Consider the familiar movie domain with the following relations. In the relations, the first argument is a numerical ID of the movies in the database.

Movie(ID, title, year, genre)
Director(ID, director)
Actor(ID, actor)

Suppose a user poses the following query, asking for comedies produced after 1950 where the director was also an actor in the movie:

$$Q(\mathsf{T},\mathsf{Y},\mathsf{D}) \text{ :- } \mathsf{Movie}(\mathsf{I},\mathsf{T},\mathsf{Y},\mathsf{G}), \mathsf{Y} \geq 1950, \mathsf{G}=\text{“comedy”}, \mathsf{Director}(\mathsf{I},\mathsf{D}), \mathsf{Actor}(\mathsf{I},\mathsf{D})$$

Suppose that, in addition, we have access to the following view over the database:

$$V_1(\mathsf{T},\mathsf{Y},\mathsf{D}) \text{ :- } \mathsf{Movie}(\mathsf{I},\mathsf{T},\mathsf{Y},\mathsf{G}), \mathsf{Y} \geq 1940, \mathsf{G}=\text{“comedy”}, \mathsf{Director}(\mathsf{I},\mathsf{D}), \mathsf{Actor}(\mathsf{I},\mathsf{D})$$

Since the selection on year in V_1 is less restrictive than in Q, we can infer that $V_1 \sqsupseteq Q$. Since the year attribute is in the head of V_1, we can use V_1 to answer Q simply by adding another selection:

$$Q'(\mathsf{T},\mathsf{Y},\mathsf{D}) \text{ :- } V_1(\mathsf{T},\mathsf{Y},\mathsf{D}), \mathsf{Y} \geq 1950$$

Answering Q using V_1 is likely to be more efficient than answering Q_1 directly from the database, because we do not need to perform the join operations in Q. However, suppose that instead of V_1 we only had access to the following views:

$$V_2(\mathsf{I},\mathsf{T},\mathsf{Y}) \text{ :- } \mathsf{Movie}(\mathsf{I},\mathsf{T},\mathsf{Y},\mathsf{G}), \mathsf{Y} \geq 1950, \mathsf{G}=\text{“comedy”}$$
$$V_3(\mathsf{I},\mathsf{D}) \text{ :- } \mathsf{Director}(\mathsf{I},\mathsf{D}), \mathsf{Actor}(\mathsf{I},\mathsf{D})$$

Neither V_2 nor V_3 contain Q, and therefore the same reasoning as above does not apply. However, we can still answer Q using V_2 and V_3 as follows, also leading to a more efficient evaluation plan in many cases:

$$Q''(\mathsf{T},\mathsf{Y},\mathsf{D}) \text{ :- } V_2(\mathsf{I},\mathsf{T},\mathsf{Y}), V_3(\mathsf{I},\mathsf{D})$$

The example above illustrates that deciding whether a query can be answered from a set of precomputed views is a more general problem than query containment, because we need to consider *combinations* of views. In data integration, such combinations will be quite important because we will consider combinations of data sources, each corresponding to a view, to answer a user query. This section describes algorithms for answering queries using views and provides a deeper understanding for when views can be used and when not.

2.4.1 Problem Definition

We begin by formally defining the problem of answering queries using views. Throughout this section we assume that we are given a set of views denoted as $V_1, \ldots, V_m$. Unless stated otherwise, we will assume that the language for specifying the query and the views is conjunctive queries.

Given a query Q and a set of view definitions $V_1, \ldots, V_m$, a rewriting of the query using the views is a query expression Q' that refers *only* to the views $V_1, \ldots, V_m$. In SQL, a query refers only to the views if all the relations mentioned in the FROM clause are views. In the notation of conjunctive queries, a query refers only to the views if all the subgoals are views with the exception of interpreted atoms. In practice, we may also be interested in rewritings that can also refer to the database relations, but all the algorithms we describe in this section can easily be extended to that case.

The most natural question to pose is whether there is an *equivalent* rewriting of the query that uses the views.

Definition 2.5 (Equivalent Query Rewritings). *Let Q be a query and $\mathcal{V} = \{V_1, \ldots, V_m\}$ be a set of view definitions. The query Q' is an equivalent rewriting of Q using $\mathcal{V}$ if*

- *Q' refers only to the views in $\mathcal{V}$, and*
- *Q' is equivalent to Q.* □

For example, the query Q'' in Example 2.14 is an equivalent rewriting of Q using V_2 and V_3. Note that when we consider the equivalence between a query Q and a rewriting Q', we actually need to consider the unfolding of Q' w.r.t. the views.

There may not always be an equivalent rewriting of the query using a set of views, and we may want to know what is the *best* we can do with a set of views. For example, suppose that instead of V_2 in Example 2.14, we had the following view that includes comedies produced after 1960:

$V_4(\mathsf{I,T,Y})$:- $\mathsf{Movie(I,T,Y,G)}$, $\mathsf{Y} \geq 1960$, G=“comedy”

While we can't use V_4 to completely answer Q_1, the following rewriting is the best we can do and may be our only choice if we do not have access to any other data:

$Q'''(\mathsf{T,Y,D})$:- $V_4(\mathsf{I,T,Y})$, $V_3(\mathsf{I,D})$

Q''' is called a *maximally contained* rewriting of Q using V_3 and V_4. Intuitively, Q''' is the best conjunctive query that uses the views to answer the query. If we also have the following view:

$V_5(\mathsf{I,T,Y})$:- $\mathsf{Movie(PI,T,Y,G)}$, $\mathsf{Y} \geq 1950$, $\mathsf{Y} \leq 1955$, G=“comedy”

then we can construct a union of conjunctive queries that is the best rewriting we can find.

The following definition makes these intuitions precise. To define maximally contained rewritings, we first need to specify the exact language we consider for rewritings (e.g., do we allow unions, interpreted predicates?). If we restrict or otherwise change the language of the rewriting, the maximally contained rewriting may change. In the definition below, the query language is denoted by $\mathcal{L}$.

Definition 2.6 (Maximally Contained Rewritings). *Let Q be a query, $\mathcal{V} = \{V_1, \ldots, V_m\}$ be a set of view definitions, and $\mathcal{L}$ be a query language. The query Q' is a maximally contained rewriting of Q using $\mathcal{V}$ w.r.t. $\mathcal{L}$ if*

- *Q' is a query in $\mathcal{L}$ that refers only to the views in $\mathcal{V}$,*
- *Q' is contained in Q, and*
- *there is no rewriting $Q_1 \in \mathcal{L}$, such that $Q' \sqsubseteq Q_1 \sqsubseteq Q$ and Q_1 is not equivalent to Q'.* □

When a rewriting Q' is contained in Q but is not a maximally contained rewriting we refer to it as a contained rewriting.

Example 2.15

If we consider the language of the rewriting to be unions of conjunctive queries, the following is the maximally contained rewriting of Q:

$Q^{(4)}(\mathsf{T},\mathsf{Y},\mathsf{D})$:- $V_4(\mathsf{I},\mathsf{T},\mathsf{Y})$, $V_3(\mathsf{I},\mathsf{D})$
$Q^{(4)}(\mathsf{T},\mathsf{Y},\mathsf{D})$:- $V_5(\mathsf{I},\mathsf{T},\mathsf{Y})$, $V_3(\mathsf{I},\mathsf{D})$

If we restrict $\mathcal{L}$ to be conjunctive queries, then either of the two rules defining $\mathsf{Q}^{(4)}$ would be a maximally contained rewriting. This example illustrates that the maximally contained rewriting need not be unique.

The algorithms for finding equivalent rewritings differ a bit from those for finding maximally contained rewritings. As we explain in Chapter 3, in the context of data integration, views describe the contents of the data sources. However, the sources may be incomplete and not contain all the data conforming to their descriptions. In addition, the sources together may not cover the entire domain of interest. In the rest of this chapter, unless otherwise mentioned, our goal is to find a union of conjunctive queries that is the maximally contained rewriting of a conjunctive query using a set of conjunctive views.

2.4.2 When Is a View Relevant to a Query?

Informally, a few conditions need to hold in order for a view to be useful for answering a query. First, the set of relations the view mentions should overlap with that of the query. Second, if the query applies predicates to attributes that it has in common with the view, then the view must apply either equivalent or logically weaker predicates in order to be part of an equivalent rewriting. If the view applies a logically stronger predicate, it may be part of a contained rewriting.

The following example illustrates some of the subtleties in answering queries using views. Specifically, it shows how small modifications to the views render them useless for answering a given query.

Example 2.16

Consider the following views:

$V_6(\mathsf{T},\mathsf{Y})$:- Movie(I, T, Y, G), Y ≥ 1950, G="comedy"
$V_7(\mathsf{I},\mathsf{T},\mathsf{Y})$:- Movie(I, T, Y, G), Y ≥ 1950, G="comedy", Award(I, W)

V_8(I, T) :- Movie(I, T, Y, G), Y $\geq$ 1940, G="comedy"

V_6 is similar to V_1, except that it does not include the ID attribute of the Movie table in the head, and therefore we cannot perform the join with V_2. The view V_7 considers only the comedies produced in or after 1950 that also won at least one award. Hence, the view applies an additional condition that does not exist in the query and cannot be used in an equivalent rewriting. Note, however, that if we have an integrity constraint stating that every movie wins an award (unlikely, unfortunately), then V_7 would actually be usable. Finally, view V_8 applies a weaker predicate on the year than in the query, but the year is not included in the head. Therefore, the rewriting cannot apply the appropriate predicate on the year.

The next few sections will give precise conditions and efficient algorithms for answering queries using views.

2.4.3 The Possible Length of a Rewriting

Before we discuss specific algorithms for rewriting queries using views, a first question we may ask is: What is the space of possible rewritings we even need to consider? We now describe a fundamental result that shows that if a conjunctive query has n subgoals, then we need not consider query rewritings that have more than n subgoals. This result enables us to focus on the set of rewritings that have at most n subgoals, and find efficient ways of searching this set.

Theorem 2.9. *Let Q and $\mathcal{V} = \{V_1, \ldots, V_m\}$ be conjunctive queries without comparison predicates or negation, and let n be the number of subgoals in Q.*

- *If there is an equivalent conjunctive rewriting of Q using $\mathcal{V}$, then there is one with at most n subgoals.*
- *If Q' is a contained conjunctive rewriting of Q using $\mathcal{V}$, and has more than n subgoals, then there is a conjunctive rewriting Q'', such that $Q \sqsupseteq Q'' \sqsupseteq Q'$, and Q'' has at most n subgoals.* □

Proof. Let the query be of the form

$$Q(\bar{A}) \text{ :- } e_1(\bar{A}_1), \ldots, e_n(A_n)$$

and suppose that

$$Q'(\bar{X}) \text{ :- } v_1(\bar{X}_1), \ldots, v_m(X_m)$$

is an equivalent rewriting of Q using $\mathcal{V}$, where the v_i's are subgoals referring to view relations, and that $m > n$. Let the unfolding of Q' w.r.t. the view definitions be

$$\begin{aligned} Q'_u(\bar{X}) \text{ :- } & e_1^1(\bar{X}_1^1), \ldots, e_1^{j_1}(\bar{X}_1^{j_1}), \\ & e_2^1(\bar{X}_2^1), \ldots, e_2^{j_2}(\bar{X}_2^{j_2}), \\ & \vdots \\ & e_m^1(\bar{X}_m^1), \ldots, e_m^{j_m}(\bar{X}_m^{j_m}) \end{aligned}$$

Note that Q'_u is equivalent to Q'. Since $Q \sqsupseteq Q'$, there must be a containment mapping from Q to Q'_u. Let the containment mapping from Q to Q'_u be ψ. Since Q has only n subgoals, the image of ψ can only be present in at most n rows of the definition of Q'_u. Let us assume without loss of generality that these are the first k rows, where $k \leq n$. Now consider the rewriting Q'' that has only the first k subgoals of Q'. The containment mapping ψ still establishes that $Q \sqsupseteq Q''$, and since Q'' has a subset of the subgoals of Q', then clearly $Q'' \sqsupseteq Q'$, and hence $Q'' \sqsupseteq Q$. Consequently, $Q \equiv Q''$, and Q'' has at most n subgoals. The proof of the second part of the theorem is almost identical. □

Theorem 2.9 yields a first naive algorithm for finding a rewriting of a query given a set of views. Specifically, there are a finite number of rewritings of length at most n subgoals using a set of views $\mathcal{V}$, because there are a finite number of variable patterns we can consider for each view. Hence, to find an equivalent rewriting, the algorithm can proceed as follows:

- Guess a rewriting Q' of Q that has at most n subgoals.
- Check if $Q' \equiv Q$.

Similarly, the union of all rewritings Q' of length at most n subgoals, such that $Q' \sqsubseteq Q$ is a maximally contained rewriting of Q.

This algorithm shows that finding an equivalent rewriting of a conjunctive query using a set of conjunctive views is in NP. In fact, it can be shown that the problem is also NP-hard, and therefore NP-complete. Employing this algorithm is impractical because it means enumerating all possible rewritings of length at most n subgoals and checking equivalence of these rewritings to the query. The next sections describe algorithms for finding a maximally contained rewriting that are efficient in practice.

2.4.4 The Bucket and MiniCon Algorithms

This section describes two algorithms, the Bucket Algorithm and the MiniCon Algorithm, that drastically reduce the number of rewritings we need to consider for a query given a set of views. Broadly speaking, the algorithms first determine a set of view atoms that are *relevant* to each subgoal, and then consider only the combinations of these view atoms. The MiniCon Algorithm goes a step further and also considers some *interactions* between view subgoals, therefore further reducing the number of combinations that need to be considered.

The Bucket Algorithm

The main idea underlying the Bucket Algorithm is that the number of query rewritings that need to be considered can be drastically reduced if we first consider each subgoal in the query in isolation, and determine which views may be relevant to that subgoal. As we discuss later, the Bucket Algorithm is not guaranteed to find all the maximally contained rewritings. However, since the reduction in the number of rewritings is best illustrated when the query and views include interpreted predicates, we use them in our algorithm description and example.

The first step of the algorithm is shown in Algorithm 3. The algorithm constructs for each subgoal g in the query a bucket of relevant view atoms. A view atom is relevant if one of its subgoals can play the role of g in the rewriting. To do that, several conditions must be satisfied: (1) the view subgoal should be over the same relation as g, (2) the interpreted predicates of the view and the query are mutually satisfiable after the appropriate substitution is made, and (3) if g includes a head variable of the query, then the corresponding variable in V must also be a head variable in the view.

Algorithm 3. CreateBuckets: Creating the buckets.

Input: conjunctive query Q of the form $Q(\bar{X}) : -R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), c_1, \ldots, c_l$; a set of conjunctive views $\mathcal{V}$. **Output:** list of buckets.

for $1 \leq i \leq n$ **do**
  Initialize $Bucket_i$ to $\emptyset$
end for
for $i = 1, \ldots, n$ **do**
  for each $V \in \mathcal{V}$ **do**
    Let V be of the form: $V(\bar{Y}) : -S_1(\bar{Y}_1), \ldots, S_m(\bar{Y}_m), d_1, \ldots, d_k$
    for $j = 1, \ldots, m$ **do**
      if $R_i = S_j$ **then**
        // Let ψ be the mapping defined on the variables of V as follows:
        Let y be the bth variable in $\bar{Y}_j$ and x be the bth variable in $\bar{X}_i$
        if $x \in \bar{X}$ and $y \notin \bar{Y}$ **then**
          ψ is undefined and move to the next j
        else if $y \in \bar{Y}$ **then**
          $\psi(y) = x$
        else
          $\psi(y)$ is a new variable that does not appear in Q or V
        end if
        // Let Q' be the query defined as follows:
        $Q' : -R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), c_1, \ldots, c_l, S_1(\psi(\bar{Y}_1)), \ldots, S_m(\psi(\bar{Y}_m)), \psi(d_1), \ldots, \psi(d_k)$
        if Q' is satisfiable **then**
          Add $\psi(V)$ to $Bucket_i$
        end if
      end if
    end for
  end for
end for
return $Bucket_1, \ldots, Bucket_n$

The second step of the Bucket Algorithm considers all the possible combinations of rewritings. Each combination that includes a view atom from each bucket (eliminating duplicate atoms if necessary). This phase of the algorithm differs depending on whether we are looking for an equivalent rewriting of the query using the views or the maximally contained rewriting:

- For an equivalent rewriting, we consider each candidate rewriting Q' and check whether $Q \equiv Q'$ or whether there are interpreted atoms C that can be added to Q' such that $Q \wedge C \equiv Q'$.
- For the maximally contained rewriting we construct a union of conjunctive queries by considering each conjunctive rewriting Q':
  - If $Q' \sqsubseteq Q$, then Q' is included in the union.
  - If there exist interpreted atoms C that can be added to Q' such that $Q' \wedge C \sqsubseteq Q$, then $Q' \wedge C$ is included in the union.
  - If $Q' \not\sqsubseteq Q$ but there exists a homomorphism, ψ, on the head variables of Q' such that $\psi(Q') \sqsubseteq Q$, then we include $\psi(Q')$ in the union.

Example 2.17

Continuing with the example from the movie domain, suppose we also have the relation Revenues(ID, Amount), describing the revenues each movie garnered over time, and suppose movie IDs are integers. Consider a query that asks for directors whose movies garnered a significant amount of money:

$$
\begin{aligned}
Q(\mathsf{ID}, \mathsf{Dir}) \text{ :- } & \mathsf{Movie}(\mathsf{ID}, \mathsf{Title}, \mathsf{Year}, \mathsf{Genre}), \mathsf{Revenues}(\mathsf{ID}, \mathsf{Amount}), \mathsf{Director}(\mathsf{ID}, \mathsf{Dir}), \\
& \mathsf{Amount} \geq \$100\mathrm{M}
\end{aligned}
$$

Suppose we have the following views:

$$
\begin{aligned}
V_1(\mathsf{I}, \mathsf{Y}) & \text{ :- } \mathsf{Movie}(\mathsf{I}, \mathsf{T}, \mathsf{Y}, \mathsf{G}), \mathsf{Revenues}(\mathsf{I}, \mathsf{A}), \mathsf{I} \geq 5000, \mathsf{A} \geq \$200\mathrm{M} \\
V_2(\mathsf{I}, \mathsf{A}) & \text{ :- } \mathsf{Movie}(\mathsf{I}, \mathsf{T}, \mathsf{Y}, \mathsf{G}), \mathsf{Revenues}(\mathsf{I}, \mathsf{A}) \\
V_3(\mathsf{I}, \mathsf{A}) & \text{ :- } \mathsf{Revenues}(\mathsf{I}, \mathsf{A}), \mathsf{A} \leq \$50\mathrm{M} \\
V_4(\mathsf{I}, \mathsf{D}, \mathsf{Y}) & \text{ :- } \mathsf{Movie}(\mathsf{I}, \mathsf{T}, \mathsf{Y}, \mathsf{G}), \mathsf{Director}(\mathsf{I}, \mathsf{D}), \mathsf{I} \leq 3000
\end{aligned}
$$

In the first step the algorithm creates a bucket for each of the relational subgoals in the query in turn. The resulting contents of the buckets are shown in Table 2.1. The bucket of Movie(ID,Title,Year,Genre) includes views V_1, V_2, and V_4. Note that each view head in a bucket only includes variables in the domain of the mapping. Fresh variables (primed) are used for the other head variables of the view (such as A′ in V_2(ID,A′)).

The bucket of the subgoal Revenues(ID,Amount) contains the views V_1 and V_2, and the bucket of Director(ID,Dir) includes an atom of V_4. There is no atom of V_3 in the bucket of Revenues(I,A) because constraint on the revenues considered in V_3 is not mutually satisfiable with the predicates in the query.

In the second step of the algorithm, we combine elements from the buckets. The first combination, involving the first element from each bucket, yields the rewriting

$$
q'_1(\mathsf{ID}, \mathsf{Dir}) \text{ :- } V_1(\mathsf{ID}, \mathsf{Year}), V_1(\mathsf{ID}, \mathsf{Y}'), V_4(\mathsf{ID}, \mathsf{Dir}, \mathsf{Y}'')
$$

Table 2.1 Contents of the Buckets*.

Movie(ID,Title,Year,Genre)	Revenues(ID,Amount)	Director(ID,Dir)
V_1(ID,Year) V_2(ID,A′) V_4(ID,D′,Year)	V_1(ID,Y′) V_2(ID,Amount)	V_4(ID,Dir,Y′)

*The primed variables are those that are not in the domain of the unifying mapping.

However, while both V_1 and V_4 are relevant to the query in *isolation*, their combination is guaranteed to be empty because they cover disjoint sets of movie identifiers.

Considering the second elements in the two left buckets yields the rewriting

$$q'_2(\mathsf{ID},\mathsf{Dir}) \text{ :- } V_2(\mathsf{ID},\mathsf{A}'),\ V_2(\mathsf{ID},\mathsf{Amount}),\ V_4(\mathsf{ID},\mathsf{Dir},\mathsf{Y}')$$

As is, this rewriting is not contained in the query, but we can add the predicate $\mathsf{Amount} \geq \$100\mathsf{M}$, and remove one redundant subgoal $V_2(\mathsf{ID},\mathsf{A}')$ to obtain a contained rewriting

$$q'_3(\mathsf{ID},\mathsf{Dir}) \text{ :- } V_2(\mathsf{ID},\mathsf{Amount}),\ V_4(\mathsf{ID},\mathsf{Dir},\mathsf{Y}'),\ \mathsf{Amount} \geq \$100\mathrm{M}$$

Finally, combining the last elements in each of the buckets yields

$$q'_4(\mathsf{ID},\mathsf{Dir}) \text{ :- } V_4(\mathsf{ID},\mathsf{D}',\mathsf{Year}),\ V_2(\mathsf{ID},\mathsf{Amount}),\ V_4(\mathsf{ID},\mathsf{Dir},\mathsf{Y}')$$

However, after removing the first subgoal, which is redundant, and adding the predicate $\mathsf{Amount} \geq \$100\mathrm{M}$, we would obtain q'_3 again, which is the only contained rewriting the algorithm finds.

The MiniCon Algorithm

The MiniCon Algorithm also proceeds in two steps. It begins like the Bucket Algorithm, considering which views contain subgoals that are relevant to a subgoal g in the query. However, once the algorithm finds a partial mapping from a subgoal g in the query to a subgoal g_1 in a view V, it tries to determine which *other* subgoals from that view must also be used in conjunction with g_1. The algorithm considers the join predicates in the query (which are specified by multiple occurrences of the same variable) and finds the minimal additional set of subgoals that need to be mapped to subgoals in V, given that g will be mapped to g_1. This set of subgoals and mapping information is called a *MiniCon Description* (MCD) and can be viewed as a generalization of buckets. The following example illustrates the intuition behind the MiniCon Algorithm and how it differs from the Bucket Algorithm.

Example 2.18

Consider the following example over the same movie schema:

$$\begin{aligned} Q_1(\mathsf{Title},\mathsf{Year},\mathsf{Dir}) &\text{ :- } \mathsf{Movie}(\mathsf{ID},\mathsf{Title},\mathsf{Year},\mathsf{Genre}),\ \mathsf{Director}(\mathsf{ID},\mathsf{Dir}),\ \mathsf{Actor}(\mathsf{ID},\mathsf{Dir}) \\ V_5(\mathsf{D},\mathsf{A}) &\text{ :- } \mathsf{Director}(\mathsf{I},\mathsf{D}),\ \mathsf{Actor}(\mathsf{I},\mathsf{A}) \end{aligned}$$

V_6(T,Y,D,A) :- Director(I,D), Actor(I,A),Movie(I,T,Y,G)

In its first step, the Bucket Algorithm would put the atoms V_5(Dir,A') and V_5(D',Dir) in the buckets of Director(ID,Dir) and Actor(ID,Dir), respectively. However, a careful analysis reveals that these two bucket items cannot contribute to a rewriting. Specifically, suppose the variable Dir is mapped to the variable D in V_5 (see figure below). The variable ID needs to be mapped to the variable I in V_5, but I is not a head variable of V_5, and therefore we will not be able to join the Director subgoal of V_5 with the other subgoals in the query.

Q_1(Title, Year, Amount) :- Director(ID, Dir),Actor(ID,Dir),Movie(ID, Title, Year, Genre)
↓ ↓
V_5(D,A) :- Director(I, D) Actor(I, A)

The MiniCon Algorithm realizes that V_5 cannot be used to answer the query and therefore ignores it. We now describe the details of the algorithm.

STEP 1: CREATING MCDs

A MCD is a mapping from a subset of the variables in the query to variables in one of the views. Intuitively, a MCD represents a fragment of a containment mapping from the query to the rewriting of the query. The way in which we construct the MCDs guarantees that these fragments can later be combined seamlessly.

In the algorithm description, we use the following terms. First, given a mapping τ from *Vars*(Q) to *Vars*(V), we say that a view subgoal g_1 *covers* a query subgoal g if $\tau(g) = g_1$. Second, we often need to consider *specializations* of a view, formed by equating some of the head variables of the view (e.g., V_6(T,Y,D,D) instead of V_6(T,Y,D,A) in our example). We describe these specializations with *head homomorphisms*. A head homomorphism h on a view V is a mapping h from *Vars*(V) to *Vars*(V) that is the identity on the existential variables, but may equate distinguished variables, i.e., for every distinguished variable x, $h(x)$ is distinguished, and $h(x) = h(h(x))$. We can now define MCDs formally.

Definition 2.7 (MiniCon Descriptions). *A MCD C for a query Q over a view V is a tuple of the form $(h_C, V(\bar{Y})_C, \varphi_C, G_C)$ where*

- *h_C is a head homomorphism on V,*
- *$V(\bar{Y})_C$ is the result of applying h_C to V, i.e., $\bar{Y} = h_C(\bar{A})$, where $\bar{A}$ are the head variables of V,*
- *φ_C is a partial mapping from Vars(Q) to h_C(Vars(V)), and*
- *G_C is a subset of the subgoals in Q that are covered by some subgoal in $h_C(V)$ using the mapping φ_C.* □

In words, φ_C is a mapping from Q to the specialization of V obtained by the head homomorphism h_C. The main idea underlying the algorithm is to carefully choose the set G_C of subgoals of Q that we cover by the mapping φ_C. The algorithm will use only the MCDs that satisfy the following property:

Property 2.1. *Let C be a MCD for Q over V. The MiniCon Algorithm considers C only if it satisfies the following conditions.*

C1. For each head variable X of Q that is in the domain of φ_C, $\varphi_C(X)$ is a head variable in $h_C(V)$.
C2. If $\varphi_C(X)$ is an existential variable in $h_C(V)$, then for every g, subgoal of Q, that includes X (1) all the variables in g are in the domain of φ_C, and (2) $\varphi_C(g) \in h_C(V)$.

Clause C1 is also required by the Bucket Algorithm. Clause C2 captures the intuition we illustrated in our example. If a variable X is part of a join predicate that is not enforced by the view, then X must be in the head of the view so the join predicate can be applied by another subgoal in the rewriting. In our example, clause C2 would rule out the use of V_5 for query Q_1.

Algorithm 4 builds the MCDs. Note that the algorithm will not consider all the possible MCDs, but only those in which h_C is the least restrictive head homomorphism necessary in order to unify subgoals of the query with subgoals in a view.

Algorithm 4. FormMCDs: First phase of the MiniCon Algorithm. Note that condition (b) minimizes G_c *given* a choice of h_C and φ_C, and is therefore not redundant with condition (c).

Input: conjunctive query Q; set of conjunctive queries $\mathcal{V}$. **Output:** set of MCDs $\mathcal{C}$.
  Initialize $\mathcal{C} := \emptyset$
  for each subgoal $g \in Q$ **do**
    for view $V \in \mathcal{V}$ and every subgoal $v \in V$ **do**
      Let h be the least restrictive head homomorphism on V such that there exists a mapping φ, s.t. $\varphi(g) = h(v)$
      if h and φ exist **then**
        Add to $\mathcal{C}$ any new MCD C that can be constructed where:
        (a) φ_C (resp. h_C) is an extension of φ (resp. h),
        (b) G_C is the minimal subset of subgoals of Q such that G_C, φ_C, and h_C satisfy Property 2.1, and
        (c) it is not possible to extend φ and h to φ'_C and h'_C s.t. (b) is satisfied and G'_C, as defined in (b), is a subset of G_C
      end if
    end for
  end for
  return $\mathcal{C}$

Example 2.19

In our example, after realizing that V_5 cannot be part of any MCD, the algorithm would create an MCD for V_6, whose components are

$$(A \rightarrow D, V_6(T, Y, D, D), Title \rightarrow T, Year \rightarrow Y, Dir \rightarrow D, \{1,2,3\})$$

Note that in the MCD the head homomorphism equates the variables D and A in V_6, and that the MCD includes all the subgoals from the query.

STEP 2: COMBINING THE MCDs

In the second phase, the MiniCon Algorithm combines the MCDs to create conjunctive rewritings and outputs a union of conjunctive queries. Because of the way in which the MCDs were constructed, the second phase of the algorithm is actually simpler and more efficient than the corresponding one in the Bucket Algorithm. Specifically, any time a set of MCDs covers mutually disjoint subsets of subgoals in the query, but together cover all the subgoals, the resulting rewriting is guaranteed to be a contained rewriting. Hence, we do not need to perform any containment checks in this phase. Algorithm 5 combines MCDs.

MINIMIZING THE REWRITINGS

The rewritings resulting from the second phase of the algorithm may be redundant. In general, they can be minimized by techniques for conjunctive query minimization. However, one case of redundant subgoals can be identified more simply as follows. Suppose a rewriting Q' includes two atoms A_1 and A_2 of the same view V, whose MCDs were C_1 and C_2, and the following conditions are satisfied: (1) whenever A_1 (resp. A_2) has a variable from Q in position i, then A_2 (resp. A_1) has either the same variable or a variable that does not appear in Q in that position, and (2) the ranges of φ_{C_1} and φ_{C_2} do not overlap on existential variables of V. In this case we can remove one of the two atoms by applying to Q' the homomorphism τ that is (1) the identity on the variables of Q and (2) the most general unifier of A_1 and A_2.

CONSTANTS IN THE QUERY AND VIEWS

When the query or the view includes constants, we make the following modifications to the algorithm. First, the domain and range of φ_C in the MCDs may also include constants. Second, a MCD also records a (possibly empty) set of mappings ψ_C from variables in $Vars(Q)$ to constants.

When the query includes constants, we add the following condition to Property 2.1:

C3. If a is a constant in Q, it must be the case that either (1) $\varphi_C(a)$ is a distinguished variable in $h_C(V)$ or (2) $\varphi_C(a)$ is the constant a.

Algorithm 5. CombineMCDs: Second phase of the MiniCon Algorithm—combining the MCDs.

Input: MCDs $\mathcal{C}$, of the form $(h_C, V(\bar{Y}), \varphi_C, G_C, EC_C)$. **Output:** rewritten query.
// Given a set of MCDs, $C_1, \ldots, C_n$, we define the function EC on $Vars(Q)$ as follows:
if for $i \neq j$, $EC_{\varphi_i}(x) \neq EC_{\varphi_j}(x)$ **then**
Define $EC_C(x)$ to be one of them arbitrarily, but consistently across all y for which $EC_{\varphi_i}(y) = EC_{\varphi_i}(x)$
end if
Initialize $Answer = \emptyset$
for every subset $C_1, \ldots, C_n$ of $\mathcal{C}$ such that (1) $G_{C_1} \cup G_{C_2} \cup \ldots \cup G_{C_n} = subgoals(Q)$ and (2) **for** every $i \neq j$, $G_{C_i} \cap G_{C_j} = \emptyset$ **do**
Define a mapping Ψ_i on the $\bar{Y}_i$'s as follows:
if there exists a variable $x \in Q$ such that $\varphi_i(x) = y$ **then**
Let $\Psi_i(y) = x$
else
Let Ψ_i be a fresh copy of y
Create the conjunctive rewriting $Q'(EC(\bar{X}))$:- $V_{C_1}(EC(\Psi_1(\bar{Y}_{C_1}))), \ldots, V_{C_n}(EC(\Psi_n(\bar{Y}_{C_n})))$
Add Q' to *Answer*
end if
end for
return *Answer*

When the views have constants, we modify Property 2.1 as follows:

- We relax clause C1: a variable x that appears in the head of the query must either be mapped to a head variable in the view (as before) or be mapped to a constant a. In the latter case, the mapping $x \to a$ is added to ψ_C.
- If $\varphi_C(x)$ is a constant a, then we add the mapping $x \to a$ to ψ_C. (Note that condition C2 only applies to existential variables, and therefore if $\varphi_C(x)$ is a constant that appears in the body of V but not in the head, a MCD is still created.)

Next, we combine MCDs with some extra care. Two MCDs, C_1 and C_2, both of which have x in their domain, can be combined only if either they (1) both map x to the same constant, or (2) one (e.g., C_1) maps x to a constant and the other (e.g., C_2) maps x to a distinguished variable in the view. Note that if C_2 maps x to an existential variable in the view, then the MiniCon Algorithm would never consider combining C_1 and C_2 in the first place, because they would have overlapping G_C sets.

Finally, we modify the definition of EC, such that whenever possible, it chooses a constant rather than a variable.

COMPUTATIONAL COMPLEXITY

The worst-case computational complexity of the Bucket and MiniCon Algorithms is the same. In both cases the running time is $O(nmM)^n$, where n is the number of subgoals in the query, m is the maximal number of subgoals in a view, and M is the number of views.

2.4.5 A Logical Approach: The Inverse-Rules Algorithm

In this section we describe an algorithm for rewriting queries using views that takes a purely logical approach to the problem. The following example illustrates the intuition behind the algorithm.

Example 2.20

Consider the view from our previous example:

V_7(I, T, Y, G) :- Movie(I, T, Y, G), Director(I, D), Actor(I, D)

Suppose we know that the tuple (79522, Manhattan, 1979, Comedy) is in the extension of V_7. Clearly, we can infer from it that Movie(79522, Manhattan, 1979, Comedy) holds. In fact, the following rule would be logically sound:

IN_1: Movie(I, T, Y, G) :- V_7(I, T, Y, G)

We can also infer from V_7(79522, Manhattan, 1979, Comedy) that some tuples exist in Director and Actor, but it's a bit trickier. In particular, we do not know which value of D in the database yielded V_7(79522, Manhattan, 1979, Comedy). All we know is that there is some constant, c, such that Director(79522,c), Actor(79522, c) hold. We can express this inference using the following rules:

IN_2: Director(I, f_1(I, T, Y, G)) :- V_7(I, T, Y, G)
IN_3: Actor(I, f_1(I, T, Y, G)) :- V_7(I, T, Y, G)

The term f_1(I,T,Y,G) is called a *Skolem term* and denotes some constant that depends on the values I, T, Y, and G and the function name f_1. Given two nonidentical Skolem terms, we do not know whether they refer to the same constant in the database or not, only that they both exist.

The query rewriting produced by the Inverse-Rules Algorithm includes all the inverse rules we can write for each of the view definitions and the rule defining the query. In our example, the inverse rules are IN_1, IN_2, and IN_3. To illustrate, suppose our query asked for all movie titles, with their genres and years:

Q_2(Title, Year, Genre) :- Movie(ID, Title, Year, Genre)

Evaluating the inverse rules over the extension V_7(79522, Manhattan, 1979, Comedy) yields the tuples

Movie(79522, Manhattan, 1979, Comedy),
Director(79522, f_1((79522, Manhattan, 1979, Comedy))
Actor(79522, f_1((79522, Manhattan, 1979, Comedy)).

Evaluating Q_2 on that set of tuples would yield the answer Movie(Manhattan, 1979, Comedy).

Algorithm 6. CreateInverseRules: Inverse rule rewriting algorithm. Note that the answer contains the original query and the inverse rules $\mathcal{R}$.

Input: conjunctive query Q; set of conjunctive view definitions $\mathcal{V}$. **Output:** datalog program to answer Q.
Let $\bar{R} = \emptyset$
for every $V \in \mathcal{V}$ **do**
  Let V be of the form: $v(\bar{X})$:- $p_1(\bar{X}_1)$, ..., $p_n(\bar{X}_n)$ and suppose the existential variables in V are $Y_1, \dots, Y_m$
  for $j = 1, \dots, n$ **do**
    LET $\bar{X}'_j$ be the tuple of variables created from $\bar{X}_j$ as follows:
    if $X = Y_k$ **then**
      X is replaced by $f_{v,k}(\bar{X})$ in $\bar{X}'_j$
    else
      X is unchanged in X'_j
    end if
    Add to $\mathcal{R}$ the inverse rule: $p_j(\bar{X}'_j)$:- $v(\bar{X})$
  end for
end for
return $Q \cup \bar{R}$

Algorithm 6 creates the inverse-rule rewriting.

2.4.6 Comparison of the Algorithms

The strength of the Bucket Algorithm is that it exploits the comparison atoms in the query to prune significantly the number of candidate conjunctive rewritings that need to be considered. Checking whether a view should belong to a bucket can be done in time polynomial in the size of the query and view definition when the predicates involved are arithmetic comparisons. Hence, if the views are indeed distinguished by having different interpreted predicates, then the resulting buckets will be relatively small. This is a common scenario when we apply answering queries using views to integrating data on the World Wide Web, where sources (modeled as views) are distinguished by the geographical area they apply to or other interpreted predicates.

However, the Bucket Algorithm does not consider interactions between different subgoals in the query and the views, and therefore the buckets may contain irrelevant subgoals. The MiniCon Algorithm overcomes that issue. Furthermore, because of the way MCDs are built, the second phase of the MiniCon Algorithm does not require a containment check, and is thereby more efficient. In experiments, the MiniCon Algorithm is uniformly faster than the Bucket Algorithm.

The main advantage of the Inverse-Rules Algorithm over the Bucket and MiniCon algorithms is its conceptual simplicity, being based on a purely logical approach to inverting the view definitions. Because of this simplicity, the algorithm can also be applied when the query Q is recursive and, as we discuss in Chapter 3, to cases where there are known functional dependencies. In addition, the inverse rules can be created in time that is polynomial in the size of the view definitions. Note that the algorithm does not tell us whether the maximally contained rewriting is equivalent to the original query (and hence avoids NP-hardness).

On the other hand, the rewriting produced by the Inverse-Rules Algorithm often leads to a query that that is more expensive to evaluate over the view extensions. The reason is that the Bucket and MiniCon Algorithms create buckets and MCDs in a way that considers the context of the query Q, while the inverse rules are computed only based on the view definition. In experiments, the MiniCon Algorithm has been shown to typically be faster than the Inverse-Rules Algorithm.

2.4.7 View-Based Query Answering

Thus far we have described algorithms for finding the maximally contained rewriting for a query given a set of views. However, these rewritings are maximal with respect to the query language we consider for the rewritings. In our discussion, we have considered rewritings that we can express as unions of conjunctive queries.

In this section we consider a more general question: Given a query Q, a set of view definitions $\mathcal{V} = V_1, \ldots, V_n$, and extensions for each of the views, $\bar{v} = v_1, \ldots, v_n$, what are all the answers to Q that can be inferred from $\mathcal{V}$ and $\bar{v}$? We begin by introducing the notion of *certain answers* that enables us to formally state the above question, and then describe some basic results on finding certain answers.

Given a database D and a query Q, we know with certainty all the answers to Q. However, if we are only given $\mathcal{V}$ and $\bar{v}$, then we do not precisely know the contents of D. Instead, we only have *partial information* about the real state of the database D. The information is partial in the sense that all we know is the answer to certain queries posed on D.

Example 2.21

Suppose we have the following views computing the set of actors and directors, respectively:

V_8(Dir) :- Director(ID, Dir)
V_9(Actor) :- Actor(ID, Actor)

Suppose we are given the following view extensions:

v_8: {Allen, Coppola}
v_9: {Keaton, Pacino}

There are multiple databases that may have led to these view extensions. For example, in one, the pairs of director and actor would be ((Allen, Keaton), (Coppola, Pacino)), while in another it could be ((Allen, Pacino), (Coppola, Keaton)). In fact, the database that contains all the possible pairs would also lead to the same view extensions.

With partial information, all we know is that D is in some set $\mathcal{D}$ of *possible* databases. However, for each database in $D \in \mathcal{D}$, the answer to Q may be different. The *certain answers*, which we define below, are the ones that are answers to Q in *every* database in $\mathcal{D}$.

Before we can formally define certain answers, we need to be explicit about our assumptions about the *completeness* of the views. We distinguish between the *closed-world assumption* and the *open-world assumption*. Under the closed-world assumption, the extensions $\bar{v}$ are assumed to contain *all* the tuples in their corresponding views, while under the open-world assumption they may contain only a subset of the tuples in the views. The closed-world assumption is the one typically made when using views for query optimization, and the open-world assumption is typically used in data integration.

We can now formally define the notion of certain answers.

Definition 2.8 (Certain Answers). *Let Q be a query and $\mathcal{V} = \{V_1, \ldots, V_m\}$ be a set of view definitions over the database schema $R_1, \ldots, R_n$. Let the sets of tuples $\bar{v} = v_1, \ldots, v_m$ be extensions of the views $V_1, \ldots, V_m$, respectively.*

The tuple $\bar{t}$ is a certain answer to the query Q under the closed-world assumption given $v_1, \ldots, v_m$ if $\bar{t} \in Q(D)$ for all database instances D such that $V_i(D) = v_i$ for every i, $1 \le i \le m$.

The tuple $\bar{t}$ is a certain answer to the query Q under the open-world assumption given $v_1, \ldots, v_m$ if $\bar{t} \in Q(D)$ for all database instances D such that $V_i(D) \supseteq v_i$ for every i, $1 \le i \le m$. □

Example 2.22

Consider the views

V_8(Dir) :- Director(ID, Dir)
V_9(Actor) :- Actor(ID, Actor)

Under the closed-world assumption, there is only a single database that is consistent with the view extensions. Hence, given the query

Q_4(Dir, Actor) :- Director(ID, Dir), Actor(ID, Actor)

the tuple (Allen, Keaton) is a certain answer. However, under the open-world assumption, this tuple is no longer a certain answer because the view extensions may be missing the other actors and directors.

Certain Answers under the Open-World Assumption

Under the open-world assumption, when the query does not contain interpreted predicates, the union of conjunctive queries produced by the MiniCon and Inverse-Rules Algorithms are guaranteed to compute all the certain answers to a conjunctive query given a set of conjunctive views. The following theorem shows that the Inverse-Rules Algorithm produces all the certain answers.

Theorem 2.10. *Let Q be a conjunctive query and let $\mathcal{V} = V_1, \ldots, V_n$ be conjunctive queries, where neither Q nor $\mathcal{V}$ contain interpreted atoms or negation. Let Q' be the result of the Inverse-Rules Algorithm on Q and $\mathcal{V}$. Given extensions $\bar{v} = v_1, \ldots, v_n$ for the views in $\mathcal{V}$, evaluating Q' over $\bar{v}$ will produce all the certain answers for Q w.r.t. $\mathcal{V}$ and $\bar{v}$.* □

Proof. Denote by $\mathcal{D}$ the set of databases that are consistent with the extensions $\bar{v}$. To prove the theorem, we need to show that if $\bar{t} \in Q(D)$ for every $D \in \mathcal{D}$, then $\bar{t}$ would be produced by evaluating Q' over $\bar{v}$.

Evaluating Q' on $\bar{v}$ can be divided into two steps. In the first, we evaluate all the inverse rules on $\bar{v}$ to produce a *canonical database* D'. In the second step, we evaluate Q on D'. The proof is based on showing that if $\bar{t}$ is a tuple of constants with no functional terms and $\bar{t} \in Q(D')$, then $\bar{t} \in Q(D)$ for every $D \in \mathcal{D}$.

Suppose $\bar{a} \in v_i$ where V_i is of the form

$$v(\bar{\mathsf{X}}) \;\text{:-}\; \mathsf{p}_1(\bar{\mathsf{X}}_1), \ldots, \mathsf{p}_m(\bar{\mathsf{X}}_m)$$

and the existential variables in V are $\mathsf{Y}_1, \ldots, \mathsf{Y}_k$. If $D \in \mathcal{D}$, then there must be constants, $b_1, \ldots, b_k$, and a variable mapping ψ that maps $\bar{\mathsf{X}}$ to $\bar{a}$ and $\mathsf{Y}_1, \ldots, \mathsf{Y}_k$ to $b_1, \ldots, b_k$, such that $\mathsf{p}_i(\psi(\bar{\mathsf{X}}_\mathsf{i})) \in D$ for $1 \le i \le n$.

Evaluating the inverse rules on $\bar{v}$ produces exactly these tuples, except instead of $b_1, \ldots, b_k$ we get $f_{v_i,1}(\bar{a}), \ldots, f_{v_i,k}(\bar{a})$. In fact, *all* the tuples in D' are produced in this way for some $\bar{a} \in v_i$, so there are no extra tuples in D' that are not required by some $v_i(\bar{a})$.

Hence, we can characterize the databases in $\mathcal{D}$ as follows. Every $D \in \mathcal{D}$ can be created from D' by mapping the functional terms in D' to constants (possibly equating two distinct functional terms) and adding more tuples. In other words, for every database $D \in \mathcal{D}$ there is a homomorphism, ϕ, such that $\phi(D) \supseteq D'$.

Consequently, it is easy to see that if $\bar{t} \in Q(D')$, then $\bar{t} \in Q(D)$ for every $D \in \mathcal{D}$, since any pair of constants that are equal in D' are guaranteed to be equal in D. □

Theorem 2.10 can be used to show that the MiniCon Algorithm produces all the certain answers under the same condition. We leave it to the reader to show that any answer produced by the Inverse-Rules Algorithm is also produced by the MiniCon Algorithm, and therefore it produces all the certain answers as well.

Corollary 2.1. *Let Q be a conjunctive query, and let $\mathcal{V} = V_1, \ldots, V_n$ be conjunctive queries. The Inverse-Rules and MiniCon Algorithms produce the maximally contained union of conjunctive query rewriting of Q using $\mathcal{V}$.* □

Certain Answers under the Closed-World Assumption

Under the closed-world assumption, finding all the certain answers turns out to be computationally much harder. The following theorem shows that finding all the certain answers is co-NP-hard in the size of the data. Hence, none of the query languages we considered for rewriting will produce all the certain answers.

Theorem 2.11. *Let Q be a query and $\mathcal{V}$ be a set of view definitions. The problem of determining, given a view instance, whether a tuple is a certain answer under the closed-world assumption is co-NP-hard.* □

Proof crux. We prove the theorem by a reduction from the 3-colorability problem. Let $G = (V, E)$ be an arbitrary graph. Consider the view definitions

v_1(X) :- color(X,Y)
v_2(Y) :- color(X,Y)
v_3(X,Y) :- edge(X,Y)

and consider the instance I with $I(v_1) = V$, $I(v_2) = \{red, green, blue\}$, and $I(v_3) = E$. It can be shown that under the closed-world assumption, the query

$q()$:- edge(X,Y), color(X,Z), color(Y,Z)

has a certain answer if and only if the graph G is not 3-colorable. Because testing a graph's 3-colorability is NP-complete, the theorem follows. Note that q is a query with arity 0, hence it has a certain answer if and only if for any database, there is at least one substitution that satisfies the body. □

Certain Answers for Queries with Interpreted Predicates

When queries and views include interpreted predicates, the results on finding certain answers or maximally contained rewritings do not all carry over. In fact, the fundamental result on the limit on the size of rewritings (Theorem 2.9) does not hold any more.

The hard case is when the query contains interpreted predicates. The following theorem shows that it suffices for the query to contain the $\neq$ predicate, and then the problem of finding all certain answers is co-NP-complete.

Theorem 2.12. *Let Q be a query and $\mathcal{V}$ be a set of view definitions, all conjunctive queries, but Q may have the $\neq$ predicate. The problem of determining, given a view instance, whether a tuple is a certain answer under the open-world assumption is co-NP-hard.* □

Proof crux. We prove the theorem by a reduction from the problem of testing satisfiability of a CNF formula. Let ψ be a CNF formula with variables $x_1, \ldots, x_n$ and conjuncts $c_1, \ldots, c_m$. Consider the following conjunctive view definitions and their corresponding view instances:

v_1(X,Y,Z) :- p(X,Y,Z)
v_2(X) :- r(X,Y)
v_3(Y) :- p(X,Y,Z), r(X,Z)

$$I(v_1) = \{(i,j,1) \mid x_i \text{ occurs in } c_j\} \cup \{(i,j,0) \mid \neg x_i \text{ occurs in } c_j\}$$
$$I(v_2) = \{(1), \ldots, (n)\}$$
$$I(v_3) = \{(1), \ldots, (m)\}$$

Finally, consider the following query:

$q()$:- r(X, Y), r(X, Y′), Y $\neq$ Y′

It is possible to show that q has a certain answer under the open-world assumption if and only if the formula ψ is not satisfiable. Since the problem of testing satisfiability is NP-complete, the theorem follows. □

Fortunately, there are two important cases where the algorithms we described still yield the maximally contained rewritings and all the certain answers:

- if the query does not contain interpreted predicates (but the views may), and
- if all the interpreted predicates in the query are semi-interval comparisons.

The proof is left as an exercise for the reader.

Bibliographic Notes

Query containment has been an active area of research for many years, beginning with Chandra and Merlin [114], who proved that query containment and equivalence are NP-complete for conjunctive queries. Sagiv and Yannakakis first considered queries with unions and negation [502]. Rarajaman and Chekuri [130] describe several cases where checking query containment can be done in time polynomial in the size of the queries. Saraiya shows that when predicates do not appear more than twice in a conjunctive query, query containment can be checked in polynomial time [505].

Containment for conjunctive queries with interpreted predicates was first considered by Klug [347] (establishing the upper bound) and van der Meyden [557] (establishing the lower bound). Afrati et al. [16] consider more efficient algorithms for testing containment with interpreted predicates. In [12], the authors present a thorough treatment of query containment and answering queries using views with arithmetic comparisons. The algorithm we describe in Section 2.3.4 and the algorithm for query containment with negated subgoals in Section 2.3.5 are from Levy and Sagiv [382]. Benedikt and Gottlob [64] show that query containment for nonrecursive datalog is complete for co-NEXPTIME. Containment for queries with bag semantics were first considered by Chaudhuri and Vardi [127], and then by [320, 332]. Cohen [142] surveys algorithms for query containment with grouping and aggregation. Green [267] describes containment results for queries over *annotated relations* that cover also queries over bag semantics.

Answering queries using views has been considered in the context of data integration, query optimization (e.g., [13, 63, 125, 259, 588]), and maintenance of physical data independence (e.g., [551, 586]). A survey of the main techniques for answering queries using views is described by Halevy [284]. The result on the length of a rewriting in Section 2.4.3 is from Levy et al. [398] and the Bucket Algorithm is a slightly modified version of the algorithm described in [381]. The MiniCon Algorithm [482] and the SVB Algorithm [439] are also complete and more efficient. In [12] the authors discuss the subtleties of answering queries using views in the presence of interpreted predicates in the query and the views, and they show why the Bucket Algorithm may not be complete in the presence of interpreted predicates. In [349] the authors describe a significant speedup compared to the MiniCon Algorithm. The Inverse-Rules Algorithm is from Duschka and Genesereth [198]. An approach for answering queries using views based on the Chase Algorithm is described in [480]. Several works (e.g., [108]) considered answering queries using views over description logics. Certain answers were introduced by Abiteboul and Duschka [6], as well as the basic complexity results we described. Libkin [389] describes a general framework for defining incomplete information in databases from which he derives several results concerning certain answers for relational data and for XML data.