Query Planning with Limited Source Capabilities

Chen Li*
Computer Science Department
Stanford University
chenli@db.stanford.edu

Edward Chang
Electrical & Computer Engineering
University of California, Santa Barbara
echang@ece.ucsb.edu

Abstract

In information-integration systems, sources may have diverse and limited query capabilities. In this paper we show that because sources have restrictions on retrieving their information, sources not mentioned in a query can contribute to the query result by providing useful bindings. In some cases we can access sources repeatedly to retrieve bindings to answer a query, and query planning thus becomes considerably more challenging. We find all the obtainable answers to a query by translating the query and source descriptions to a simple recursive Datalog program, and evaluating the program on the source relations. This program often accesses sources that are not in the query. Some of these accesses are essential, as they provide bindings that let us query sources, which we could not do otherwise. However, some of these accesses can be proven not to add anything to the query's answer. We show in which cases these off-query accesses are useless, and prove that in these cases we can compute the complete answer to the query by using only the sources in the query. In the cases where off-query accesses are necessary, we propose an algorithm for finding all the useful sources for a query. We thus solve the optimization problem of eliminating the unnecessary source accesses, and optimize the program to answer the query.

Keywords: *information-integration systems, limited source capabilities, Datalog programs.*

1. Introduction

The rapid growth of the Internet is giving us access to an unprecedented number of heterogeneous information sources. Many information-integration systems (e.g., TSIMMIS [4], the Information Manifold [14], Garlic [21], Infomaster [9], Disco [22], Tukwila [12], and InfoSleuth [2]) have been proposed to support seamless access to these heterogeneous data sources. To perform queries on these sources, many studies [6, 13, 19, 20] construct answers to queries using views. These approaches are closely related to query-containment algorithms for conjunctive queries and for Datalog programs [24].

In heterogeneous environments, especially in the context of the World Wide Web, sources may have diverse and limited query capabilities. For example, many Web bookstores like `amazon.com` [1] and `barnesandnoble.com` [3] provide some search forms. A user fills out a form by specifying the values of attributes, e.g., book title, author name, publisher, and ISBN, so that the source returns the books satisfying the query conditions. These sources do not accept queries such as "return the information about all the books you know about." There are many reasons for the source restrictions, including the concerns of efficiency and security, and the limitations of the source interfaces.

In this paper, we consider a practical information-integration problem: *querying sources with limited capabilities*. We first show that because sources have restrictions on retrieving their information, sources not directly mentioned in a query can contribute to the query result, as shown by the following example.

EXAMPLE 1.1 Assume that we want to compare the average prices of the books sold by `amazon.com` and `barnesandnoble.com`. Since both sources require each source query to specify at least one value of ISBN, author, or title, we cannot retrieve all their information about books. On the other hand, although we may have some known book information such as some authors, book titles, and ISBNs, the available information may not be enough to sample the two sources. However, suppose that we can access the source `prenhall.com` to retrieve the authors of books published by Prentice Hall. We use these authors to query `amazon.com` and `barnesandnoble.com`. After retrieving the books, we can compare the average prices of the two sources.[1] □

*Research partially supported by the Stanford Graduate Fellowship and the National Science Foundation under grant #IRI-9631952.

[1] We use this example to illustrate the idea of obtaining bindings from

Example 1.1 suggests that we can use the source `prenhall.com` to retrieve bindings for the author domain to answer the query, although this source is not mentioned directly in the query. In some cases, as shown by the motivating example in Section 2, we can access sources repeatedly to obtain bindings to compute more results to a query. In Section 3 we propose a framework of query planning in integration systems with source restrictions. In the framework, source descriptions and a query are translated into a Datalog program, and we compute the *maximal* answer to the query by evaluating the program on the source relations. Datalog is used in the query planning since the planning process can be recursive, although the query itself is not.

Being able to obtain the maximal results is desirable. However, the challenge is to return the results with the minimum cost. In other words, we do not want to involve all sources blindly during the plan-generation process. In Section 4 we show that in some cases a query does not need any bindings from off-query sources. In these cases we prove that the complete answer to the query can be computed by using only the sources in the query. In the cases where it is necessary to access off-query sources, we show in Section 5 that not all the sources that contribute bindings to the query are really necessary. We thus want to include judiciously only those sources that provide some values at a place where they are needed. We develop an algorithm for finding all the useful sources for a query. We solve the optimization problem of eliminating the unnecessary source accesses, and optimize the program to answer the query (Section 6).

We discuss in Section 7 how to explore other possibilities for obtaining bindings, e.g., by using cached data and domain knowledge. In the cases where a user may be interested in a partial answer to a query, we do not need to compute the maximal answer, which may be expensive to retrieve. We discuss how to compute a partial answer to a query, and the tradeoff between the number of results and the cost of an execution plan.

In this study we focus on a class of *connection queries*. A connection query is a natural join of distinct source views with the necessary selection and projection. (The details are described in Section 2.) Here we take the following universal-relation-like assumption [23]: different attributes sharing the same name in different views have the same meaning. However, universal-relation study did not consider restrictions of retrieving information from relations. In addition, as we will see in Section 2.2, a connection query can be generated in general cases, where our techniques are applicable.

Here is a summary of the contributions of this study:

1. We show that in information-integration systems, sources not in a query can contribute to the query result because of source restrictions. In some cases, we can obtain bindings by accessing sources repeatedly to answer a query, thus query planning in the presence of restrictions becomes considerably more challenging.
2. We propose a query-planning framework, in which source descriptions and a query are translated into a Datalog program. We evaluate the program on the source relations to compute the maximal obtainable answer to the query.
3. We show how to decide whether accessing off-query sources is necessary. In the cases where it is not necessary, we prove the complete answer to the query can be computed by using only the sources in the query.
4. In the cases where we need other sources to contribute bindings, we propose an algorithm for finding the useful sources and constructing an efficient program to compute the answer.

1.1. Related work

There are two approaches to information integration [6]:

1. The source-centric approach: Both user queries and source views are in terms of some global views. For each query, the integration system needs to plan how to answer the query using source views. The Information Manifold and Infomaster follow this approach.
2. The query-centric approach: User queries are in terms of views synthesized at a mediator [26] that are defined on source views. After view expansion [16] at the mediator, the query is translated to a logical plan that is composed of the source views. TSIMMIS follows this approach, and we follow this approach in this paper.

Ullman [24] gave a good survey on the differences between these two approaches. Many studies have been done by taking the source-centric approach. For example, Qian [19] discussed how to use query folding to rewrite queries using views without considering source restrictions. Rajaraman, Sagiv, and Ullman [20] proposed algorithms for answering queries using views with binding patterns. Duschka and Levy [7] considered source restrictions by translating source binding patterns into rules in a Datalog program, assuming that all attributes share one domain. The paper did not discuss how to trim useless sources, thus it may generate programs that are not efficient to evaluate.

By taking the query-centric approach, [16] showed how to generate an executable plan of a query based on source restrictions. If the complete answer to the query cannot

sources not mentioned in the query. If we consider the possibility that bookstores and publishers may make deals, and therefore prices at a given bookseller for books by a given publisher may be atypical, a better strategy would be: use the authors from `prenhall.com` to retrieve book titles from the two bookstore sources, then use these titles to retrieve more authors. After several iterations, we average the prices of the books that are not published by Prentice Hall.

be retrieved, [16] would not answer the query, but would claim that an executable plan does not exist. In this case, our approach can still compute a partial answer. Although we take the query-centric approach in this study, our techniques for finding useful sources are also applicable to the source-centric approach, since when source views are the same as global predicates, the query-centric approach in [7] and our framework generate equivalent Datalog programs. Other related studies include how to optimize conjunctive queries with source restrictions [8, 28], how to describe source capabilities using a powerful language [25], how to compute mediator capabilities given source capabilities [27], and how to convert data at mediators [5].

2. Preliminaries

In this section, we present our motivating example and introduce the notation used in the paper.

EXAMPLE 2.1 Assume that we are building a system that integrates the information from four sources of musical CDs, as shown in Table 1. Sources s_1 and s_2 have information about CDs and their songs; sources s_3 and s_4 have information about CDs, their artists, and their prices. To simplify the notation, we use attribute *Song* for song title and attribute *Cd* for CD title. The "Must Bind" column in the table indicates the attributes that must be specified at a source. For instance, every query sent to s_2 must provide a CD title. In other words, without the information about CD titles, source s_2 cannot be queried to produce answers.

Source	Contents	Must Bind
s_1	$\mathtt{v_1(Song, Cd)}$	Song
s_2	$\mathtt{v_2(Song, Cd)}$	Cd
s_3	$\mathtt{v_3(Cd, Artist, Price)}$	Cd
s_4	$\mathtt{v_4(Cd, Artist, Price)}$	Artist

Table 1. Four sources of musical CDs

Source-view schemas can be represented by a hypergraph [23], in which each node is an attribute and each hyperedge is a source view. The hypergraph of the four views is shown in Figure 1, which also shows the tuples at each source. To simply the presentation, we use symbols t_i, c_j, and a_k to represent a song title, CD, and artist, respectively. For instance, the source view $v_1(Song, Cd)$ contains two tuples: $<t_1, c_1>$ and $<t_2, c_3>$. The figure shows the adornments of the attributes in each view: b means that the attribute must be bound, f means the attribute can be free.

Suppose a user wants to find the prices of the CDs that contain a song titled t_1. The answer can be obtained by taking the union of the following four joins: $v_1 \bowtie v_3$, $v_1 \bowtie v_4$, $v_2 \bowtie v_3$, and $v_2 \bowtie v_4$, and performing a selection $Song = t_1$ and then a projection onto the attribute

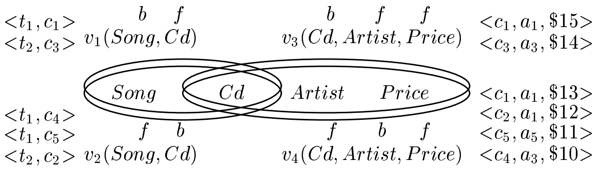

Figure 1. The hypergraph representation

Price. Figure 1 shows that there are four CDs containing the song: $<c_1, a_1, \$15>$, $<c_1, a_1, \$13>$, $<c_5, a_5, \$11>$, and $<c_4, a_3, \$10>$. Therefore, without considering the source restrictions, the answer is $\{\$15, \$13, \$11, \$10\}$. However, due to the limited source capabilities, only the \$15 can be computed if we process each join in the query at one time (as in [10, 14, 16]). The reason is that, $v_1 \bowtie v_3$ yields the \$15 in the answer; $v_1 \bowtie v_4$ cannot be executed by using only v_1 and v_4, since v_4 requires that attribute $Artist$ be specified, but we cannot bind this attribute using only these two views. Similarly, neither of the other two joins can be executed. As a consequence, the user misses the cheaper source for CD c_1 and entirely misses CDs c_4 and c_5. □

In this study, we first propose a framework that can retrieve more results from sources with restrictions. Instead of considering each join individually, our framework involves other sources not in a join to produce bindings to answer the join. For instance, when joining v_1 and v_4, we also consider the information provided by v_2 and v_3. As we will see in Section 3.3, our framework can find two additional CDs containing the song titled t_1: $<c_1, a_1, \$13>$ and $<c_4, a_3, \$10>$. If the user wants to find the cheapest CD, our approach can save \$5 for the user!

2.1. Source views

Now we give the notation used throughout the paper. Let an integration system have n sources, say, $s_1, \ldots, s_n$. Assume that each source s_i provides its data in the form of a relational view v_i. If sources have other data models, we can use *wrappers* [11] to create the simple relational view of data. In the case where one source has multiple relations, we can represent this source with multiple logical sources each of which exports only one relational view.

We assume that differences in ontologies, vocabularies, and formats used by sources have been resolved. In particular, if two sources share an attribute name, we assume that the attributes are equivalent (i.e., wrappers take care of any differences). Related research [17, 18] suggests ways to deal with ontology and format differences. We assume that the schemas of the source views are defined on a global set of attributes. Each view schema is a list of global attributes, and different views may share the same schema. For instance, in Example 2.1, we have four global attributes:

$Song$, $Title$, $Artist$, and $Price$; views v_1 and v_2 share the same schema $(Song, Cd)$.

The query capability of each source is described as a template with a binding pattern [23] representing the possible query forms that the source can accept. The adornments for the attributes in the binding pattern include b (the attribute must be bound) and f (the attribute can be free). For simplicity of exposition, we assume that each view has one template. We use v_i to stand for both the source view and its adorned template, and we believe the distinction should be clear in context. Let $\mathcal{A}(v_i)$ denote the attributes in a source view v_i, and let $\mathcal{B}(v_i)$ and $\mathcal{F}(v_i)$ be the sets of bound and free attributes in the adorned template of v_i, respectively. For instance, in Example 2.1, $\mathcal{B}(v_1) = \{Song\}$, $\mathcal{F}(v_1) = \{Cd\}$, and $\mathcal{A}(v_1) = \{Song, Cd\}$. Let $\mathcal{V}$ denote the source views with their adornments, $\mathcal{A}(\mathcal{V})$ be the attributes in $\mathcal{V}$, and $\mathcal{R}$ be an instance of the source relations of $\mathcal{V}$.

2.2. Queries

A user query is represented in the form

$$\mathcal{Q} = <\mathcal{I}, O, \mathcal{C}>$$

where $\mathcal{I}$ is a list of input assignments of the form `attribute = constant`, O is a list of output attributes whose values the user is interested in, and $\mathcal{C}$ is a list of *connections*. Each connection is a set of source views that connect the input attributes and the output attributes. As we will see shortly, we interpret a connection as the natural join of the views in the connection. The following are some possible ways in which $\mathcal{C}$ could be generated:

1. It is generated by query expansion at a mediator, as in TSIMMIS [16].
2. It is generated by a minimal-connection algorithm, as in universal-relation systems [23].
3. It is specified explicitly by the user.

For instance, the query in Example 2.1 can be represented as

$$\mathcal{Q} = <\{Song = t_1\}, \{Price\}, \{T_1, T_2, T_3, T_4\}>$$

in which the four connections are: $T_1 = \{v_1, v_3\}$, $T_2 = \{v_1, v_4\}$, $T_3 = \{v_2, v_3\}$, and $T_4 = \{v_2, v_4\}$. Note that there can be multiple input attributes and multiple output attributes in a query. Let $I(\mathcal{Q})$ and $O(\mathcal{Q})$ respectively denote the input attributes and the output attributes of query $\mathcal{Q}$. $I(\mathcal{Q})$ and $O(\mathcal{Q})$ do not overlap. Let $\mathcal{A}(T)$ be all the attributes in a connection T.

2.3. The answer to a query

Suppose T is a connection in query $\mathcal{Q}$. For those tuples in the *natural join* of the relations in T that satisfy the input constraints in $\mathcal{Q}$, their projections onto the output attributes are the *complete answer for connection T*. The union of the answers for all the connections in $\mathcal{Q}$ is the *complete answer to query $\mathcal{Q}$*. Due to the limited source capabilities, the *obtainable answer to a query* is the *maximal* answer to the query that can be retrieved from the sources, using only the initial bindings in the query and the source relations.

The complete answer to a user query could be retrieved if the sources did not have limited capabilities. However, we may get only a partial answer to the query due to the source restrictions. For instance, in Example 2.1, the complete answer to the query is $\{\$15, \$13, \$11, \$10\}$, while as we will see in Section 3.3, the obtainable answer is $\{\$15, \$13, \$10\}$. Given source descriptions and a query, if the complete answer to the query cannot be computed, our framework collects as much information as possible to answer the query. In the rest of this paper, unless otherwise specified, the term *the answer for a connection* means the obtainable answer for the connection, and the *answer to a query* is the union of the obtainable answers for all the connections in the query.

3. Query planning

In this section, we propose a framework of query planning in the presence of source restrictions. In the framework source descriptions and a query are translated into a Datalog program, which can be evaluated to answer the query. We also discuss the efficiency of the program.

3.1. Constructing the program $\Pi(\mathcal{Q}, \mathcal{V})$

Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, we translate them into a Datalog program, denoted $\Pi(\mathcal{Q}, \mathcal{V})$, that incorporates the source restrictions and the query, and thus can be evaluated on the source relations. For instance, Figure 2 shows the Datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ for the query and the source views in Example 2.1. We use names beginning with lower-case letters for constants and predicate names, and names beginning with upper-case letters for variables. Note that this program is recursive, although query $\mathcal{Q}$ is not.

Let us look at the details of how the program $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed. For each source view v_i, we introduce an EDB predicate ([23]) v_i and an IDB predicate $\widehat{v_i}$ (called the *α-predicate* of v_i). Predicate v_i represents all the tuples at source s_i, and $\widehat{v_i}$ represents the *obtainable* tuples at s_i. Introduce a goal predicate ans to store the answer to the query; the arguments of ans correspond to the output attributes $O(\mathcal{Q})$ in $\mathcal{Q}$.

$$
\begin{array}{lll}
r_1: & \mathtt{ans(P)} & \text{:- } \widehat{\mathtt{v_1}}(\mathtt{t_1,C}), \widehat{\mathtt{v_3}}(\mathtt{C,A,P}) \\
r_2: & \mathtt{ans(P)} & \text{:- } \widehat{\mathtt{v_1}}(\mathtt{t_1,C}), \widehat{\mathtt{v_4}}(\mathtt{C,A,P}) \\
r_3: & \mathtt{ans(P)} & \text{:- } \widehat{\mathtt{v_2}}(\mathtt{t_1,C}), \widehat{\mathtt{v_3}}(\mathtt{C,A,P}) \\
r_4: & \mathtt{ans(P)} & \text{:- } \widehat{\mathtt{v_2}}(\mathtt{t_1,C}), \widehat{\mathtt{v_4}}(\mathtt{C,A,P}) \\
r_5: & \widehat{\mathtt{v_1}}(\mathtt{S,C}) & \text{:- } \mathtt{song(S)}, \mathtt{v_1(S,C)} \\
r_6: & \mathtt{cd(C)} & \text{:- } \mathtt{song(S)}, \mathtt{v_1(S,C)} \\
r_7: & \widehat{\mathtt{v_2}}(\mathtt{S,C}) & \text{:- } \mathtt{cd(C)}, \mathtt{v_2(S,C)} \\
r_8: & \mathtt{song(S)} & \text{:- } \mathtt{cd(C)}, \mathtt{v_2(S,C)} \\
r_9: & \widehat{\mathtt{v_3}}(\mathtt{C,A,P}) & \text{:- } \mathtt{cd(C)}, \mathtt{v_3(C,A,P)} \\
r_{10}: & \mathtt{artist(A)} & \text{:- } \mathtt{cd(C)}, \mathtt{v_3(C,A,P)} \\
r_{11}: & \mathtt{price(P)} & \text{:- } \mathtt{cd(C)}, \mathtt{v_3(C,A,P)} \\
r_{12}: & \widehat{\mathtt{v_4}}(\mathtt{C,A,P}) & \text{:- } \mathtt{artist(A)}, \mathtt{v_4(C,A,P)} \\
r_{13}: & \mathtt{cd(C)} & \text{:- } \mathtt{artist(A)}, \mathtt{v_4(C,A,P)} \\
r_{14}: & \mathtt{price(P)} & \text{:- } \mathtt{artist(A)}, \mathtt{v_4(C,A,P)} \\
r_{15}: & \mathtt{song(t_1)} & \text{:- }
\end{array}
$$

Figure 2. Program $\Pi(\mathcal{Q}, \mathcal{V})$ in Example 2.1

Let $T = \{v_1, \ldots, v_k\}$ be a connection in $\mathcal{Q}$. The following rule is the *connection rule* of T:

$$ans(O(\mathcal{Q})) \text{ :- } \widehat{v_1}(\mathcal{A}(v_1)), \ldots, \widehat{v_k}(\mathcal{A}(v_k))$$

where the arguments in predicate ans are the corresponding attributes in $O(\mathcal{Q})$. For each argument in $\widehat{v_i}$, if the corresponding attribute in view v_i is an input attribute of $\mathcal{Q}$, this argument is replaced by the initial value of the attribute in $\mathcal{Q}$. Otherwise, a variable corresponding to the attribute name is used as an argument in predicate $\widehat{v_i}$. For instance, in Figure 2, rules r_1, r_2, r_3, and r_4 are the connection rules of the connections T_1, T_2, T_3, and T_4, respectively.

Decide the domains of all the attributes in the views, and group the attributes into sets while the attributes in each set share the same domain. Introduce a unary *domain predicate* for each domain to represent all its possible values that can be deduced.[2] In Figure 2, the predicates $song$, cd, $artist$, and $price$ represent the domains of song titles, CD titles, artists, and prices, respectively.

Suppose that source view v_i has m attributes, say $A_1, \ldots, A_m$. Assume the adornment of v_i says that the arguments in positions $1, \ldots, p$ need to be bound, and the arguments in positions $p+1, \ldots, m$ can be free. The following rule is the *α-rule* of v_i:

$$\widehat{v_i}(A_1, \ldots, A_m) \text{ :- } domA_1(A_1), \ldots, domA_p(A_p), v_i(A_1, \ldots, A_m)$$

in which each $domA_j$ ($j = 1, \ldots, p$) is the domain predicate for attribute A_j. For $k = p+1, \ldots, l$, the following rule is a *domain rule* of v_i:

$$domA_k(A_k) \text{ :- } domA_1(A_1), \ldots, domA_p(A_p), v_i(A_1, \ldots, A_m)$$

[2]The idea of domain predicates is borrowed from [7]. However, in our framework, different domains have different domain predicates, while in [7] only one domain predicate was used for all attributes.

For instance, rule r_9 in Figure 2 is the α-rule of v_3; rules r_{10} and r_{11} are its domain rules. Assume that $A_i = a_i$ is in the assignment list $\mathcal{I}$ of $\mathcal{Q}$, the following rule is a *fact rule* of attribute A_i:

$$domA_i(a_i) \text{ :-}$$

For instance, rule r_{15} in Figure 2 is a fact rule of attribute $Song$, since we know from the query that t_1 is a song title.

The program $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed in three steps:

1. Write the connection rule for each connection in $\mathcal{Q}$.
2. Write the α-rule and the domain rules for each source view in $\mathcal{V}$.
3. Write the fact rule for each input attribute in $\mathcal{Q}$.

In Figure 2, rules r_1, r_2, r_3, and r_4 are the connection rules of T_1, T_2, T_3, and T_4, respectively. Rule r_5 is the α-rule of v_1, and r_6 is the domain rule of v_1; rules r_7 to r_{14} are the α-rules and the domain rules of the other three source views. Finally, r_{15} is the fact rule of the attribute $Song$. Recall that the views in each connection link the input attributes and the output attributes in the query. Based on how program $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed, we have the following proposition:

Proposition 3.1 *Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, the Datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ is safe ([23]).* □

3.2. Binding assumptions

During the construction of the program $\Pi(\mathcal{Q}, \mathcal{V})$, we make the following important assumptions:

1. Each binding for an attribute must be from the domain of this attribute;
2. If a source view requires a value, say, a string, as a particular argument, we will not allow the strategy of trying all the possible strings to "test" the source;
3. Rather we assume that any binding is either obtained from the user query, or from a tuple returned by another source query.

We use Example 2.1 to explain these assumptions. The first assumption says that we would not use an artist name as a binding for attribute $Song$. Notice that if two attributes have the same *type*, they can still be from two different domains. For example, the attributes $Song$ and Cd share the same *string* type, but they have two different domains.

View $v_3(Cd, Artist, Price)$ requires that a query to source s_3 give a CD title. The second assumption says that we would *not* allow the following naive "strategy": generate all possible strings to test whether s_3 has CDs with these strings as titles. This approach would not terminate, since there will be an infinite number of strings that need to be tested. The third assumption says that each bound value of an attribute A must either be derived from the user query, or be a value of A in a tuple returned by another source query. For instance, if c_1 is a CD title returned from source s_1, and $Cd = c_2$ is an initial binding in a query, then we know that c_1 and c_2 are two CD titles, and they can be used to query source s_3. In Section 7 we will discuss other possibilities for obtaining bindings.

3.3. Evaluating the program $\Pi(\mathcal{Q}, \mathcal{V})$

We evaluate the Datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ on the source relations to compute the facts for predicate ans. Note that the v_i's are the only EDB predicates in $\Pi(\mathcal{Q}, \mathcal{V})$. However, in an integration system, we do not know the tuples at each source before sending source queries. Now we show how to evaluate $\Pi(\mathcal{Q}, \mathcal{V})$ to answer the query.

To evaluate the domain rules and the α-rule of a source view v_i, predicate v_i is "populated" by source queries to s_i. Suppose that the right-hand side of its domain rules and its α-rule is:

$$domA_1(A_1), \ldots, domA_p(A_p), v_i(A_1, \ldots, A_m)$$

Once we know that $(a_1, \ldots, a_p)$ are the values of the $domA_j$'s ($j = 1, \ldots, p$), respectively, we can send a query $v_i(a_1, \ldots, a_p, A_{p+1}, \ldots, A_m)$ to source s_i. This source query is guaranteed to be executable, since it satisfies the binding requirements of v_i. The results of this source query add more tuples to the predicate $\widehat{v_i}$ (for the α-rule) and the predicates $domA_j$'s (for the domain rules).

After the evaluation of the program terminates, the facts for the domain predicates include *all* the obtainable values of these domains. Similarly, the α-predicate facts are *all* the obtainable tuples at the sources. Since $\Pi(\mathcal{Q}, \mathcal{V})$ includes the connection rules for the connections in query $\mathcal{Q}$, the facts for the goal predicate ans form the maximal obtainable answer to $\mathcal{Q}$. Thus, we have the following proposition:

Proposition 3.2 *Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, for any source relations $\mathcal{R}$ of $\mathcal{V}$, if we evaluate $\Pi(\mathcal{Q}, \mathcal{V})$ on $\mathcal{R}$, the set of facts for the predicate ans is the obtainable answer to $\mathcal{Q}$.* □

Table 2 shows how to evaluate the program in Figure 2 to compute the answer to the query in Example 2.1, and Table 3 shows the results. As expected, the program computes all the obtainable values of song titles, CD titles, artists, and prices from the four sources and the query, as well as all the obtainable tuples at the sources. The set of ans facts is the answer to the query. Therefore, our approach returns two more tuples, \$13 and \$10, than the approach in Section 2. Note that we cannot retrieve the tuple $<t_1, c_5>$ of v_2 and the tuple $<c_5, a_5, \$11>$ of v_4, since we cannot retrieve the value a_5 for attribute *Artist*, no matter what legal source queries we execute.

Order	Source Query	Returned Tuple(s)	New Bindings(s)
1	$v_1(t_1, C)$	$<t_1, c_1>$	$Cd = c_1$
2	$v_3(c_1, A, P)$	$<c_1, a_1, \$15>$	$Artist = a_1$
3	$v_4(C, a_1, P)$	$<c_1, a_1, \$13>$, $<c_2, a_1, \$12>$	$Cd = c_2$
4	$v_2(S, c_2)$	$<t_2, c_2>$	$Song = t_2$
5	$v_1(t_2, C)$	$<t_2, c_3>$	$Cd = c_3$
6	$v_3(c_3, A, P)$	$<c_3, a_3, \$14>$	$Artist = a_3$
7	$v_4(C, a_3, P)$	$<c_4, a_3, \$10>$	$Cd = c_4$
8	$v_2(S, c_4)$	$<t_1, c_4>$	

Table 2. Evaluating the program in Figure 2

IDBs	Results	IDBs	Results
$\widehat{v_1}$	$<t_1, c_1><t_2, c_3>$	$song$	t_1, t_2
$\widehat{v_2}$	$<t_1, c_4><t_2, c_2>$	cd	c_1, c_2, c_3, c_4
$\widehat{v_3}$	$<c_1, a_1, \$15><c_3, a_3, \$14>$	$artist$	a_1, a_3
$\widehat{v_4}$	$<c_1, a_1, \$13><c_2, a_1, \$12>$ $<c_4, a_3, \$10>$	$price$	\$15, \$14, \$13, \$12, \$10
ans	\$15, \$13, \$10		

Table 3. Results of the program in Figure 2

The program $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed in a brute-force way, and it needs to be optimized. In particular, for each connection T in the query, the program may access views that are not in T. However, some of these off-connection accesses do not add anything to the query's answer. In the next two sections we discuss how to decide whether accessing off-connection views is necessary, and if necessary, what source views should be accessed.

4. Accessing off-connection views

In this section we discuss how to decide whether accessing off-connection views is necessary to compute the answer for a connection. In the case where it is not necessary, we prove that the complete answer for the connection can be computed by using only the sources in the connection.

The following example shows that accessing all the views not mentioned in a connection is not always necessary to compute its maximal obtainable answer.

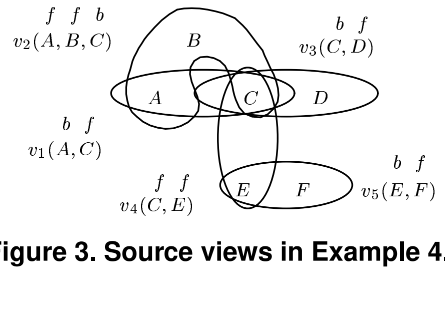

Figure 3. Source views in Example 4.1

EXAMPLE 4.1 Consider the five views in Figure 3. Suppose that a user submits a query

$$\mathcal{Q} = <\{A = a_0\}, \{D\}, \{T_1, T_2\}>,$$

which has two connections $T_1 = \{v_1, v_3\}$, $T_2 = \{v_2, v_3\}$. That is, the user knows that the value of A is a_0, and wants to get the associated D values using $v_1 \bowtie v_3$ and $v_2 \bowtie v_3$. Assume that different attributes have different domains. The corresponding Datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ is shown in Figure 4.

r_1: $\mathtt{ans(D)}$:- $\widehat{\mathtt{v_1}}(\mathtt{a_0, C}), \widehat{\mathtt{v_3}}(\mathtt{C, D})$
r_2: $\mathtt{ans(D)}$:- $\widehat{\mathtt{v_2}}(\mathtt{a_0, B, C}), \widehat{\mathtt{v_3}}(\mathtt{C, D})$
r_3: $\widehat{\mathtt{v_1}}(\mathtt{A, C})$:- $\mathtt{domA(A)}, \mathtt{v_1(A, C)}$
r_4: $\mathtt{domC(C)}$:- $\mathtt{domA(A)}, \mathtt{v_1(A, C)}$
r_5: $\widehat{\mathtt{v_2}}(\mathtt{A, B, C})$:- $\mathtt{domC(C)}, \mathtt{v_2(A, B, C)}$
r_6: $\mathtt{domA(A)}$:- $\mathtt{domC(C)}, \mathtt{v_2(A, B, C)}$
r_7: $\mathtt{domB(B)}$:- $\mathtt{domC(C)}, \mathtt{v_2(A, B, C)}$
r_8: $\widehat{\mathtt{v_3}}(\mathtt{C, D})$:- $\mathtt{domC(C)}, \mathtt{v_3(C, D)}$
r_9: $\mathtt{domD(D)}$:- $\mathtt{domC(C)}, \mathtt{v_3(C, D)}$
r_{10}: $\widehat{\mathtt{v_4}}(\mathtt{C, E})$:- $\mathtt{v_4(C, E)}$
r_{11}: $\mathtt{domC(C)}$:- $\mathtt{v_4(C, E)}$
r_{12}: $\mathtt{domE(E)}$:- $\mathtt{v_4(C, E)}$
r_{13}: $\widehat{\mathtt{v_5}}(\mathtt{E, F})$:- $\mathtt{domE(E)}, \mathtt{v_5(E, F)}$
r_{14}: $\mathtt{domF(F)}$:- $\mathtt{domE(E)}, \mathtt{v_5(E, F)}$
r_{15}: $\mathtt{domA(a_0)}$:-

Figure 4. Program $\Pi(\mathcal{Q}, \mathcal{V})$ in Example 4.1

Consider connection T_1. The program $\Pi(\mathcal{Q}, \mathcal{V})$ accesses the three views that are not in T_1 during the evaluation of the program. However, these off-connection accesses do not contribute to T_1's results. The reason is that, suppose $t = <d>$ is a tuple in the complete answer for T_1, and t comes from tuple $t_1 = <a_0, c>$ of v_1 and tuple $t_3 = <c, d>$ of v_3. By sending a query $v_1(a_0, C)$ to s_1 we can retrieve tuple t_1. With the new binding $C = c$ we can send a query $v_3(c, D)$ to s_3, and retrieve tuple t_3. Therefore, by using only the views in T_1 we can compute its complete answer.

Consider connection T_2. Since we cannot get any binding for attribute C by using only the two views in T_2, we need v_2 and v_4 to contribute C bindings. Thus these two off-connection views are useful to T_2. On the other hand, $v_5(E, F)$ does not contribute to T_2's results, because the E and F bindings from s_5 do not help obtain more answers for T_2. □

In general, given a connection T in a query $\mathcal{Q}$, we need to decide whether accessing the views outside T is necessary. Before giving the solution, we first introduce some notation.

4.1. Forward-closure

Definition 4.1 (**forward-closure**) Given a set of source views $\mathcal{W} \subseteq \mathcal{V}$ and a set of attributes $X \subseteq \mathcal{A}(\mathcal{V})$, the *forward-closure of* X *given* $\mathcal{W}$, denoted *f-closure*$(X, \mathcal{W})$, is a set of the source views in $\mathcal{W}$ such that, starting from the attributes in X as the initial bindings, the binding requirements of these source views are satisfied by using only the source views in $\mathcal{W}$. □

In other words, *f-closure*$(X, \mathcal{W})$ can be computed as follows: At the beginning, only the attributes in X are bound, and *f-closure*$(X, \mathcal{W})$ is empty. At each step, for each source view $v \in \mathcal{W} -$ *f-closure*$(X, \mathcal{W})$, check whether $\mathcal{B}(v)$, the bound attributes of v, is a subset of the bound attributes so far. If so, add v to *f-closure*$(X, \mathcal{W})$, and each attribute in $\mathcal{F}(v)$, the free attributes of v, becomes bound. Repeat this process until no more source views can be added to *f-closure*$(X, \mathcal{W})$. Let $\mathcal{A}\big($*f-closure*$(X, \mathcal{W})\big)$ denote all the attributes of the source views in *f-closure*$(X, \mathcal{W})$. Therefore, $\mathcal{A}($*f-closure*$(X, \mathcal{W}))$ includes all the attributes that can be bound eventually by using the source views in $\mathcal{W}$ starting from the initial bindings in X.

EXAMPLE 4.2 In Example 4.1:

$$\textit{f-closure}(\{A\}, \{v_1, v_2, v_3\}) = \{v_1, v_2, v_3\},$$

since we can use the bound attribute A to get tuples of v_1 and bind C, which is the only bound attribute of v_2 and v_3. In Example 2.1, *f-closure*$(\{Song\}, \{v_1, v_4\}) = \{v_1\}$, and *f-closure*$(\{Song\}, \{v_1, v_3\}) = \{v_1, v_3\}$. □

4.2. Independent connections

A connection T in a query $\mathcal{Q}$ is *independent* if

$$\textit{f-closure}(I(\mathcal{Q}), T) = T.$$

That is, the binding requirements of the source views in the connection can be satisfied by using only these source views starting from the initial bindings in $I(\mathcal{Q})$. In other words, if connection $T = \{w_1, \ldots, w_k\}$ is independent, then there exists an *executable sequence* of all the source views in connection T: $w_{i_1}, \ldots, w_{i_k}$, such that $\mathcal{B}(w_{i_1}) \subseteq I(\mathcal{Q})$, and for $j = 2, \ldots, k$, $\mathcal{B}(w_{i_j}) \subseteq I(\mathcal{Q}) \cup \mathcal{A}(w_{i_1}) \cup \cdots \cup \mathcal{A}(w_{i_{j-1}})$. That is, the binding requirements of each source view in the sequence can be satisfied by the initial bindings in $\mathcal{Q}$ and the previous source views. For instance, the connection $T_1 = \{v_1, v_3\}$ in Example 4.1 is independent, since it has an executable sequence: v_1, v_3. The following theorem shows that an independent connection does not require bindings from views outside the connection. (Due to space limitations, we have not provided all the proofs of the lemmas and theorems in this paper. They are available in the full version of the paper [15].)

Theorem 4.1 *If connection* T *is independent, then for any source relations, we can compute the complete answer for* T *by using only the source views in* T. □

Theorem 4.2 *For a nonindependent connection T, there exists an instance of the source relations, such that some tuples in the complete answer for T cannot be obtained.* $\square$

Many related studies (e.g., [8, 16]) consider the case where a connection in a query is independent. If the connection is not independent, their algorithms give up attempting to answer the connection. However, our framework can still compute a partial answer for the connection by accessing off-connection views. In the next section, we will discuss how to decide what source views need to be accessed.

5. Finding relevant views of a connection

If a connection T is not independent, we may get more bindings by accessing views not in T. Some of the off-connection accesses are actually essential, as they provide bindings that let us query the sources in the connection, which we could not do otherwise. However, some of these off-connection accesses can be proven not to add anything to the query's results. In this section, we discuss how to eliminate the unnecessary view accesses. We first give the formal definition of relevant source views of a connection, and then propose an algorithm for finding all the relevant source views of a connection. To simplify the presentation, in the rest of the paper we assume that different attributes are from different domains.

5.1. Relevant source views of a connection

Given source descriptions $\mathcal{V}$, a query $\mathcal{Q}$, and a connection T in $\mathcal{Q}$, a source view $v \in \mathcal{V}$ is *relevant* to connection T if for some source relations, removing v from $\mathcal{V}$ can change the obtainable answer for connection T; otherwise, v is *irrelevant* to connection T. In other words, a source view v is relevant to a connection T if we can miss some answers for T if we do not use v. Note that whether a source view is relevant to a connection does not depend on other connections in the query. In Example 4.1, the relevant source views of connection T_1 are the two views in T_1, while the relevant source views of connection T_2 are v_1, v_2, v_3, and v_4.

Given source descriptions $\mathcal{V}$, a query $\mathcal{Q}$, and a connection T in $\mathcal{Q}$, we need to solve the following problem: how to find all the relevant source views of T? The following example shows the challenge of this problem: not all the views that contribute bindings to T are relevant to T.

EXAMPLE 5.1 Consider the five views in Figure 5. Suppose that a user submits a query

$$\mathcal{Q} = <\{A = a\}, \{F, G\}, \{T\}>,$$

which has one connection $T = \{v_1, v_2, v_3\}$. That is, the user knows the value of A is a, and wants to get the associated F and G values using $v_1 \bowtie v_2 \bowtie v_3$. Connection T is not independent, since we cannot bind attributes D and E by using only the views in T starting from the initial binding in $\mathcal{Q}$. We need other views to bind D and E, so that we can query s_2 and s_3 to retrieve tuples. Thus, v_4 and v_5 may be useful.

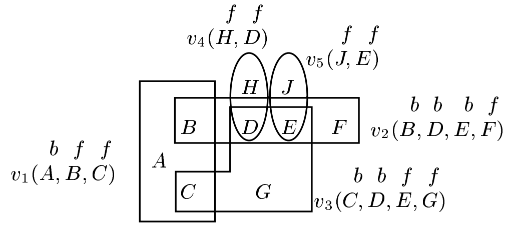

Figure 5. The source views in Example 5.1

However, although view v_5 can bind attribute E, it is *not relevant* to connection T. To illustrate the reason, we prove that using only v_1, v_2, v_3, and v_4, we can compute all the obtainable answer for T. Suppose tuple $t = <f, g>$ is in the obtainable answer, and t comes from tuple $t_1 = <a, b, c>$ of v_1, tuple $t_2 = <b, d, e, f>$ of v_2, and tuple $t_3 = <c, d, e, g>$ of v_3. Since the initial value of A in the query is a, we can send a source query $v_1(a, B, C)$ to retrieve tuple t_1 from v_1. Because attribute D is not in $I(\mathcal{Q})$, and only v_4 (with binding pattern ff) takes D as a free attribute, the value d of D must be derived from the result of a source query to s_4, which includes a tuple whose D value is d. With $C = c$ and $D = d$, we can retrieve tuple t_3 from v_3 by sending a source query $v_3(c, d, E, G)$, and then retrieve tuple t_2 from v_2 by sending a source query $v_2(b, d, e, F)$. Thus, without using v_5, we can get tuple t in the obtainable answer for connection T. The proof also shows that without using v_4, we cannot get any answer for T. $\square$

As there may be many views with different schemas and binding patterns, it becomes challenging to decide what views can really contribute to the results of a connection. Before giving the algorithm for finding all the relevant views of a connection, we require a series of definitions.

5.2. Queryable source views

A source view is *queryable* if it is in *f-closure*$(I(\mathcal{Q}), \mathcal{V})$. All the queryable source views are those that we may eventually query, starting from the initial bindings in $I(\mathcal{Q})$, and perhaps using several preliminary queries to other sources in order to get the bindings we need for these source views. Let $\mathcal{V}_q$ denote all the queryable source views in $\mathcal{V}$, and $\mathcal{A}(\mathcal{V}_q)$ be all the attributes in $\mathcal{V}_q$.

We cannot get any tuples from a nonqueryable source view, no matter what the source relations are. If a connection contains a nonqueryable source view, we cannot get any answer for this connection. Thus we need to consider only the *queryable connections* in $\mathcal{Q}$, i.e., the connections that do not have any nonqueryable source view. Clearly an independent connection is also a queryable connection, but not vice versa. For instance, in Example 4.1, connection T_2 is queryable, since both v_2 and v_3 are queryable source views, but T_2 is not independent.

5.3. Kernel, BF-chain, and backward-closure

Definition 5.1 (**kernel**) Assume T is a queryable connection in query $\mathcal{Q}$. A set of attributes $\mathcal{K} \subseteq \mathcal{A}(T)$ is a *kernel* of T if

$$\textit{f-closure}(\mathcal{K} \cup I(\mathcal{Q}), T) = T$$

and

$$\textit{f-closure}((\mathcal{K} - \{A\}) \cup I(\mathcal{Q}), T) \neq T$$

for any attribute $A \in \mathcal{K}$. □

Intuitively, a kernel $\mathcal{K}$ of connection T is a minimal set of attributes in $\mathcal{A}(T)$ such that, if the attributes in $\mathcal{K}$ have been bound, together with the initial bindings in $I(\mathcal{Q})$, we can bind all the attributes $\mathcal{A}(T)$ by using only the source views in T. In Example 4.1, $\{C\}$ is a kernel of connection T_2, because *f-closure*$(\{C\} \cup I(\mathcal{Q}), T_2) =$ *f-closure*$(\{C, A\}, T_2) = T_2$. In Example 5.1, $\{D\}$ is a kernel of the connection T, while $\{D, E\}$ is not. Since a kernel of a connection must be minimal, it cannot share any attribute with $I(\mathcal{Q})$.

We compute a kernel of a connection T by shrinking the set of attributes $X = \mathcal{A}(T) - I(\mathcal{Q})$ as much as possible while X satisfies: *f-closure*$(X \cup I(\mathcal{Q}), T) = T$. When X cannot be smaller, it will be a kernel of T. An independent connection has only one kernel: the empty set. A nonindependent connection has only nonempty kernels. It may have multiple kernels, as shown by the following example.

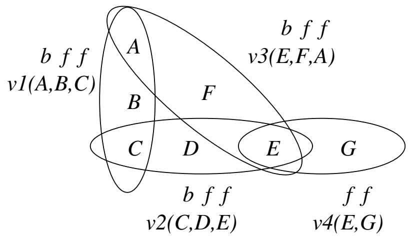

Figure 6. Multiple kernels of a connection

EXAMPLE 5.2 Figure 6 shows a hypergraph of four source views. The binding patterns for $v_1(A, B, C)$, $v_2(C, D, E)$, and $v_3(E, F, A)$ are all bff, and the binding pattern for $v_4(E, G)$ is ff. Assume a user query is $\mathcal{Q} = <\{B = b_0\}, \{A, C, E\}, \{T\}>$, in which the only connection is $T = \{v_1, v_2, v_3\}$. T has three kernels: $\{A\}$, $\{C\}$, and $\{E\}$. For instance, $\{A\}$ is a kernel because *f-closure*$(\{A\} \cup I(\mathcal{Q}), T) =$ *f-closure*$(\{A, B\}, \{v_1, v_2, v_3\}) = \{v_1, v_2, v_3\} = T$. □

Definition 5.2 (**BF-chain**) A sequence of queryable source views $w_1, \ldots, w_k$ (i.e., each $w_i \in \mathcal{V}_q$) forms a *BF-chain* (bound-free chain) if for $i = 1, \ldots, k-1$, $\mathcal{F}(w_i) \cap \mathcal{B}(w_{i+1})$ is not empty. The source views w_1 and w_k are the *head* and the *tail* of the BF-chain, respectively. □

In other words, for every two adjacent views in a BF-chain, the free attributes of the first one overlap with the bound attributes of the second, and thus the first view contributes bindings to the second one. In Example 4.1, (v_4, v_2, v_1, v_3) is a BF-chain, in which v_4 is the head and v_3 is the tail.

Definition 5.3 (**backward-closure**) Suppose that A is an attribute in $\mathcal{A}(\mathcal{V}_q)$. The *backward-closure of* A, denoted *b-closure*(A), is the set of queryable source views that can be backtracked from A by following some BF-chain in a reverse order, in which A is a free attribute of the tail in each BF-chain. □

More precisely, *b-closure*(A) can be computed as follows: Start by setting *b-closure*(A) to those source views in $\mathcal{V}_q$ that take A as a free attribute. For each view $v \in \mathcal{V}_q -$ *b-closure*(A), if there is a view $w \in$ *b-closure*(A) such that $\mathcal{F}(v)$ and $\mathcal{B}(w)$ overlap, then add v to *b-closure*(A). Repeat this process until no more queryable source views can be added to *b-closure*(A). In Example 4.1, the backward-closure of attribute C is $\{v_1, v_2, v_4\}$. The backward-closure of a set of attributes $X \subseteq \mathcal{A}(\mathcal{V}_q)$, denoted *b-closure*$(X)$, is the union of all the backward-closures of the attributes in X, i.e., *b-closure*$(X) = \bigcup_{A \in X}$ *b-closure*(A).

By the definitions of kernel, BF-chain, and backward-closure, we have the following lemmas.

Lemma 5.1 *If $\mathcal{K}$ is a kernel of a queryable connection T and A is an attribute in $\mathcal{K}$, then A is not in $\mathcal{A}$(f-closure$((\mathcal{K} - \{A\}) \cup I(\mathcal{Q}), T)$). That is, starting from the attributes of $(\mathcal{K} - \{A\}) \cup I(\mathcal{Q})$ as the initial bindings, we cannot bind attribute A by using only the source views in T.* □

Lemma 5.2 *If A_1 and A_2 are two attributes, and there is a BF-chain such that A_1 is a bound attribute of the head and A_2 is a free attribute of the tail, then b-closure$(A_1) \subseteq$ b-closure(A_2).* □

Lemma 5.3 *If connection T has two different kernels $\mathcal{K}_1$, $\mathcal{K}_2$, then b-closure$(\mathcal{K}_1) =$ b-closure$(\mathcal{K}_2)$.* □

Lemma 5.3 shows that if a connection has multiple kernels, then the backward-closures of all these kernels are the same. For instance, in Example 5.2, the connection $T = \{v_1, v_2, v_3\}$ has three kernels: $\{A\}$, $\{C\}$, and $\{E\}$, and they have the same backward-closure: $\{v_1, v_2, v_3, v_4\}$.

5.4. The algorithm FIND_REL

Now we show how to find all the relevant views of a connection by giving the following theorem:

Theorem 5.1 *If $\mathcal{K}$ is a kernel of a queryable connection T, then b-closure($\mathcal{K}$) $\cup$ T are all the relevant source views of connection T.* □

Proof: Refer to [15] for the detailed proof. The essential idea is that we need to prove, for a kernel $\mathcal{K}$ of a queryable connection T: (i) All the source views in $\mathcal{V}-$*b-closure*$(\mathcal{K}) \cup T$ are irrelevant to T; (ii) Every source view in T is relevant to T; (iii) Every source view in *b-closure*$(\mathcal{K})$ is relevant to T. We prove (i) by showing that we can get all the tuples in the obtainable answer for T by using only the source views in *b-closure*$(\mathcal{K}) \cup T$. We prove (ii) by constructing an instance of the source relations such that the obtainable answer for T is not empty, while if we remove any source view in T, the obtainable answer for T becomes empty. To prove (iii), for every source view v_i in *b-closure*$(\mathcal{K})$, we prove that v_i is relevant to T by constructing an instance of the source relations, such that without using v_i, we will miss a tuple in the obtainable answer for T. ■

Using Theorem 5.1, for a queryable connection T, we can find all its relevant source views by computing *b-closure*$(\mathcal{K}) \cup T$. If T is independent, then it has only one kernel, the empty set, whose backward-closure is empty. Thus only the source views in T are relevant to T, and this claim is consistent with Theorem 4.1. If the connection is not independent, we find a kernel $\mathcal{K}$ of T, and compute the backward-closure *b-closure*$(\mathcal{K})$. Then we find the relevant source views of T by taking the union of *b-closure*$(\mathcal{K})$ and T. Note that the backward-closures of different attributes in the kernel may overlap, and they may also overlap with the source views in T. We give an algorithm FIND_REL, as shown in Figure 7, that finds all the relevant source views of a queryable connection in a query.

Algorithm FIND_REL: Find the relevant views of a connection.
Input: • $\mathcal{V}$: Source views with binding restrictions;
• $\mathcal{Q}$: A query;
• T: A queryable connection in $\mathcal{Q}$.
Output: All the relevant views of T.
Method:
(1) Compute all the queryable source views $\mathcal{V}_q =$ *f-closure*$(I(\mathcal{Q}), \mathcal{V})$.
(2) Compute a kernel $\mathcal{K}$ of connection T;
(3) Compute the backward-closure *b-closure*$(\mathcal{K})$;
(4) Return *b-closure*$(\mathcal{K}) \cup T$.

Figure 7. The algorithm FIND_REL

EXAMPLE 5.3 In Example 4.1, all the five source views are queryable. Connection $T_1 = \{v_1, v_3\}$ is independent, so the only relevant source views of T_1 are v_1 and v_3. Connection $T_2 = \{v_2, v_3\}$ is not independent, and it has only one kernel: $\{C\}$. The backward-closure of the kernel is $\{v_1, v_2, v_4\}$, so only v_1, v_2, v_3, and v_4 are relevant to T_2.

In Example 5.1, connection $T = \{v_1, v_2, v_3\}$ has one kernel $\{D\}$, whose backward-closure is $\{v_4\}$. Thus the relevant source views of the connection are v_1, v_2, v_3, and v_4. The connection in Example 5.2 has three kernels: $\{A\}$, $\{C\}$, and $\{E\}$. We choose one of them, say $\{A\}$, and compute its backward-closure, which is $\{v_1, v_2, v_3, v_4\}$. Thus all the four views are relevant to the connection. □

Let us analyze the complexity of the algorithm FIND_REL. Suppose that there are n source views in $\mathcal{V}$. Consider a queryable connection T with m source views and k attributes. Assume it takes $O(1)$ time to check whether a set of attributes is a subset of another set of attributes. As described in Section 5.2, we can get all the queryable source views by computing *f-closure*$(I(\mathcal{Q}), \mathcal{V})$. Step 1 thus can be done in $O(n^2)$ time. Step 2 can be done by following the approach described in Section 5.3, which shrinks the attributes in $\mathcal{A}(T) - I(\mathcal{Q})$ as much as possible. Since for each set of attributes $X \subseteq \mathcal{A}(T) - I(\mathcal{Q})$, it takes $O(m^2)$ time to compute *f-closure*$(X \cup I(\mathcal{Q}), T)$, step 2 can be done in $O(km^2)$ time.

In step 3, for each attribute A in a kernel $\mathcal{K}$ of T, *b-closure*(A) can be computed in $O(n^2)$ time because during the computation, we can keep a set of attributes $\mathcal{A}_b$ as the union of the $\mathcal{B}(w_i)$'s for each w_i in *b-closure*(A) that has been computed so far. At each step, for each queryable source view v that is not in the current *b-closure*(A), we check whether $\mathcal{F}(v) \cap \mathcal{A}_b$ is not empty. If so, v is added to *b-closure*(A). Thus step 3 can be done in $O(kn^2)$ time. Therefore, the total time complexity of finding the relevant source views of the connection is $O(n^2) + O(km^2) + O(kn^2) = O(k(m^2 + n^2)) = O(kn^2)$.

6. Constructing an efficient program

In this section we show how to use the algorithm FIND_REL to construct a more efficient program than that constructed by the algorithm in Section 3, and show how to remove some useless rules in the program.

Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, we first find the relevant views of all the connections in $\mathcal{Q}$ as follows:

1. Compute all the queryable source views $\mathcal{V}_q =$ *f-closure*$(I(\mathcal{Q}), \mathcal{V})$;
2. Remove the nonqueryable connections, i.e., the connections that have a nonqueryable view;
3. Compute the relevant views for each queryable connection by calling the algorithm FIND_REL;
4. Take the union of all these relevant source views.

We then use only these relevant source views (denoted $\mathcal{V}_r$) of query $\mathcal{Q}$ to construct a Datalog program $\Pi(\mathcal{Q}, \mathcal{V}_r)$ in the same way as $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed. For instance, in Example 4.1, all the five source views are queryable. By calling the algorithm FIND_REL we find that views v_1 and v_3 are relevant to connection T_1; views v_1, v_2, v_3, and v_4 are relevant to connection T_2. Therefore, the relevant views for both connections are v_1, v_2, v_3, and v_4. We use these four views to construct a more efficient program, which can be obtained by dropping the rules r_{13} and r_{14} in Figure 4.

In addition, some useless rules in the program $\Pi(\mathcal{Q}, \mathcal{V}_r)$ can be removed, since they do not contribute to the answer. For instance, in Example 4.1, the user is not interested in the B and E values, so rules r_7 and r_{12} in Figure 4 can be dropped. Rules r_9 and r_{10} can also be removed since the predicates in their heads are not used by other rules. Figure 8 shows the optimized program that can compute the same answer as before.

$$
\begin{array}{lll}
r_1: & \mathtt{ans(D)} & \text{:- } \widehat{\mathtt{v_1}}(\mathtt{a_0, C}), \widehat{\mathtt{v_3}}(\mathtt{C, D}) \\
r_2: & \mathtt{ans(D)} & \text{:- } \widehat{\mathtt{v_2}}(\mathtt{a_0, B, C}), \widehat{\mathtt{v_3}}(\mathtt{C, D}) \\
r_3: & \widehat{\mathtt{v_1}}(\mathtt{A, C}) & \text{:- } \mathtt{domA(A)}, \mathtt{v_1(A, C)} \\
r_4: & \mathtt{domC(C)} & \text{:- } \mathtt{domA(A)}, \mathtt{v_1(A, C)} \\
r_5: & \widehat{\mathtt{v_2}}(\mathtt{A, B, C}) & \text{:- } \mathtt{domC(C)}, \mathtt{v_2(A, B, C)} \\
r_6: & \mathtt{domA(A)} & \text{:- } \mathtt{domC(C)}, \mathtt{v_2(A, B, C)} \\
r_8: & \widehat{\mathtt{v_3}}(\mathtt{C, D}) & \text{:- } \mathtt{domC(C)}, \mathtt{v_3(C, D)} \\
r_{11}: & \mathtt{domC(C)} & \text{:- } \mathtt{v_4(C, E)} \\
r_{15}: & \mathtt{domA(a_0)} & \text{:-}
\end{array}
$$

Figure 8. Optimized program in Example 4.1

In general, the useless rules in $\Pi(\mathcal{Q}, \mathcal{V}_r)$ can be found as follows: Scan through all the rules in the program $\Pi(\mathcal{Q}, \mathcal{V}_r)$, except for the connection rules. For each rule r, check whether the IDB predicate in its head is used by other rules in the program. If not, rule r is useless and can be removed from the program. Repeat this process until no useless rules can be found in the program.

7. Discussions

In this section we explore other possibilities for obtaining bindings during the query planning of a query. We also discuss how to answer a query when the user is interested in a partial answer, not necessarily the maximal answer.

7.1. Obtaining bindings

Theorem 4.1 suggests that accessing off-connection views is only necessary for nonindependent connections. So far, we have assumed that the bindings of a domain are either from a user query or from other source queries. If cached data are available, they can be incorporated into the program $\Pi(\mathcal{Q}, \mathcal{V})$ for a query $\mathcal{Q}$ and source descriptions $\mathcal{V}$. Suppose that we have a cached tuple $t_i(a_1, \ldots, a_n)$ for source view $v_i(A_1, \ldots, A_n)$. The following rules are added to the program $\Pi(\mathcal{Q}, \mathcal{V})$:

$$
\begin{array}{lll}
\widehat{v_1}(a_1, \ldots, a_n) & \text{:-} & \\
domA_i(a_i) & \text{:-} & (i = 1, \ldots, n)
\end{array}
$$

The predicates $domA_1, \ldots, domA_n$ are the domain predicates for the attributes $A_1, \ldots, A_n$, respectively. The first rule says that tuple $t_i(a_1, \ldots, a_n)$ is an obtained tuple of source view v_i. The other fact rules represent the bindings for the corresponding domains. The new rules can contribute more answers to the query. Some views that were nonqueryable when we considered only the initial bindings in $\mathcal{Q}$ may now become queryable with the new bindings from the cached data. In general, if we have some information about a domain, we can always incorporate the information into the program $\Pi(\mathcal{Q}, \mathcal{V})$ by adding the corresponding fact rules.

We may also obtain bindings by using some known domain knowledge. For example, suppose that we have a source view $student(name, dept, GPA)$ with the binding pattern *bbf*. That is, every query to this source must supply a name and a department of a student, so that the student's GPA can be returned. Assume we know that all the students at the source are in four departments: {*CS, EE, Physics, Chemistry*}. Then we can use these four departments as bindings for attribute *dept* to query the source, and we do not need other sources to contribute *dept* bindings.

7.2. Computing a partial answer

In some cases a user may be interested in a partial answer to a query. Thus we do not need to compute the maximal answer, which may be expensive to retrieve. Theorem 4.1 suggests that if a connection is independent, its complete answer can be computed by using only the views in the query. If a connection T is not independent, we can find a kernel $\mathcal{K}$ of T. We access some sources in *b-closure*($\mathcal{K}$) to obtain bindings for the attributes in $\mathcal{K}$, and compute a partial answer for the connection. Notice that we may access only a subset of the backward-closure of $\mathcal{K}$, since we are not interested in the maximal answer for T. In addition, we need to consider the tradeoff between the number of results and the number of source accesses. The more sources we access, the more bindings we can retrieve, and the more answers we can compute for the connection. We decide how many source queries to send based on how many results the user is interested in.

8. Conclusion

In information-integration systems, especially in the context of the World Wide Web, sources may have restrictions on retrieving their information. We need to consider the source restrictions while answering a user query, since we may not be able to retrieve all the data from the sources. In this paper we showed that sources not directly mentioned in a query can contribute to the query result by providing useful bindings to the query. In some cases we can access sources repeatedly to compute more results to the query. We proposed a framework of query planning in the presence of source restrictions. In the framework, a user query and source descriptions are translated into a Datalog program, and we evaluate the program on the source relations to compute the maximal obtainable answer to the query. Our framework supports recursive query planning because of the expressive power of Datalog. In addition, we showed that accessing off-query sources is not always necessary. In the case where off-query accesses do not contribute to the query results, we proved that the complete answer to the query can be computed by using only the views in the query. In the case where off-query accesses are necessary, we gave an algorithm for finding all the relevant sources to the query. Using this algorithm we can trim the unnecessary view accesses and construct an efficient Datalog program to compute the answer.

Acknowledgments: We thank Jeff Ullman for his valuable comments on this material.

References

[1] Amazon.com. `http://www.amazon.com/`.

[2] R. J. Bayardo, Jr. et al. Infosleuth: Semantic integration of information in open and dynamic environments (experience paper). *ACM SIGMOD Conference*, pages 195–206, 1997.

[3] Bn.com. `http://www.barnesandnoble.com/`.

[4] S. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *IPSJ*, pages 7–18, 1994.

[5] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! *ACM SIGMOD Conference*, pages 177–188, 1998.

[6] O. M. Duschka. Query planning and optimization in information integration. *Thesis, Computer Science Department, Stanford University*, 1997.

[7] O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI-97*, 1997.

[8] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. *ACM SIGMOD Conference*, pages 311–322, 1999.

[9] M. R. Genesereth, A. M. Keller, and O. M. Duschka. Infomaster: An information integration system. *ACM SIGMOD Conference*, pages 539–542, 1997.

[10] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. *Very Large Data Bases (VLDB) Conference*, pages 276–285, 1997.

[11] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. M. Breunig, and V. Vassalos. Template-based wrappers in the TSIMMIS system. *ACM SIGMOD Conference*, pages 532–535, 1997.

[12] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution engine for data integration. *ACM SIGMOD Conference*, pages 299–310, 1999.

[13] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. *ACM Symposium on Principles of Database Systems (PODS)*, pages 95–104, 1995.

[14] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. *Very Large Data Bases (VLDB) Conference*, pages 251–262, 1996.

[15] C. Li and E. Chang. Query planning with limited source capabilities (extended version). *Technical report, Computer Science Dept., Stanford Univ.*, 1999.

[16] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. D. Ullman, and M. Valiveti. Capability based mediation in TSIMMIS. *ACM SIGMOD Conference*, pages 564–566, 1998.

[17] D. A. Maluf and G. Wiederhold. Abstraction of representation for interoperation. *International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 441–455, 1997.

[18] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. *International Conference on Data Engineering (ICDE)*, pages 251–260, 1995.

[19] X. Qian. Query folding. *International Conference on Data Engineering (ICDE)*, pages 48–55, 1996.

[20] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. *ACM Symposium on Principles of Database Systems (PODS)*, pages 105–112, 1995.

[21] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. *Very Large Data Bases (VLDB) Conference*, pages 266–275, 1997.

[22] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[23] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes II: The New Technologies*. Computer Science Press, New York, 1989.

[24] J. D. Ullman. Information integration using logical views. *International Conference on Database Theory (ICDT)*, pages 19–40, 1997.

[25] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. *Very Large Data Bases (VLDB) Conference*, pages 256–265, 1997.

[26] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[27] R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman. Computing capabilities of mediators. *ACM SIGMOD Conference*, pages 443–454, 1999.

[28] R. Yerneni, C. Li, J. D. Ullman, and H. Garcia-Molina. Optimizing large join queries in mediation systems. *International Conference on Database Theory (ICDT)*, pages 348–364, 1999.