Optimization of Query Plans in the presence of Access Limitations

Andrea Calì, Diego Calvanese, and Davide Martinenghi

Free University of Bozen-Bolzano
Faculty of Computer Science
Piazza Domenicani, 3
I-39100 Bolzano, Italy
lastname@inf.unibz.it

Abstract. We consider the problem of querying data sources that have limited capabilities and can thus only be accessed by complying with certain binding patterns for their attributes. This is often the case, e.g., in the context of data on the web queryable via web forms as well as in legacy data wrapped in relational tables. In such contexts, computing the answer to a user query cannot be done as in a traditional database; instead, a query plan is needed that takes the access limitations into account.

In this paper, we develop a technique for producing a (possibly recursive) Datalog program that retrieves all obtainable answers for a query with limited source capabilities. In particular, we improve with respect to a previously published algorithm for optimizing query answering for conjunctive queries. Furthermore, we extend it to the context of unions of conjunctive queries. The algorithm exploits the structure of the query together with the binding patterns present in the source schema in order to compute an optimized query plan. The optimization excludes from the query plan the sources that are not relevant for the answer thus reducing the number of accesses to the sources.

1 Introduction

In the context of integration of data over the Web [3], data are often accessible only via forms, where typically certain fields are required to be filled in by the user in order to obtain a result. Consider for example the web site of an online bookstore, where, in order to obtain a list of publications extracted from an underlying database, either an author or a title has to be specified, while it is not possible to ask directly, e.g., for all books of a certain publisher. Similarly, in legacy systems where data are scattered over several files, such data may be wrapped and masked as relational tables; however, such tables typically cannot be queried freely, since there may be access limitations due to the way the data are organized in the files.

Limitations on how sources can be accessed significantly complicate query processing [11, 7, 4, 2], since in this case the query plan may fail to comply with the access limitations. As shown in [11, 7, 9, 8], query answering in the presence of access limitations in general requires the evaluation of a recursive query plan, which can be suitably expressed in Datalog. The related problem of query containment in the presence of access limitations is addressed in [10].

Since source accesses are costly, an important issue is how to reduce the number of accesses to the sources while still obtaining all possible answers to a query. Several optimizations that can be made at compile time, during query plan generation, are discussed in [7, 9, 8]. However, the presented techniques are not applicable in the case where user queries and view definitions are arbitrary conjunctive queries. In particular, in [8], the presented optimization only considers a class of queries that is a proper subset of the class of union of conjunctive queries and that is not comparable with the class of conjunctive queries; there, the problem of optimizing query answering when the query over the sources is an arbitrary conjunctive query is left open.

In [6], the author addresses the issue of *stability*, i.e., determining whether the complete answer to a query (the one that would be obtained with no access limitations) can always be computed despite the access limitations.

Another relevant topic tightly connected with this research is the problem of run-time query plan optimization in the case where constraints over sources are asserted. In [1], functional and simple full-width inclusion dependencies have been addressed, and it is shown that the implication problem for such dependencies is decidable in polynomial time. There, a necessary and sufficient condition is presented that determines, given the dependencies, whether during query evaluation a planned access to a source is necessary for computing the answer.

In this paper we address the problem of query plan optimization for sources with limited capabilities for *conjunctive queries* and for *unions of conjunctive queries*. We present a technique to optimize a query plan at the time of its generation. Such an optimization technique exploits the knowledge about the structure of the query and the binding patterns of the sources to compute a query plan that eliminates dependencies between sources and thus avoids unnecessary accesses that may be performed at query plan execution time. Moreover, this allows one to exclude from the query plan those sources that cannot contribute to the result of the query.

The rest of the paper is organized as follows. In Section 2 we introduce the technical preliminaries. In Section 3 we present a technique to construct an optimized query plan for answering conjunctive queries in the presence of access limitations. We extend this technique to unions of conjunctive queries in Section 4. We conclude in Section 5, where we also indicate possible future directions of research.

2 Preliminaries

We consider relations as sets of tuples of constants belonging to given domains and accessible via given binding patterns. Instead of using concrete domains, such as `Integer` or `String`, we deal with *abstract domains*, which have an underlying concrete domain, but represent information at a higher level of abstraction, which distinguishes, e.g., strings representing person names from strings representing plate numbers. A *relational schema* is a signature of the form $r(A_1^{(b|f)}, \ldots, A_n^{(b|f)})$, where r is the relation name, n is called the *arity* of the relation, each A_i is an abstract domain[1], and the superscript, called the *binding*, indicates that the corresponding position in the relation is *bound*, if the binding is b, or *free*, if it is f; the sequence of bindings for positions $1, \ldots, n$ in the schema is called the *binding pattern* for r. In the following we will only indicate b bindings and will consider f as the default. A *relation* over a relational schema $r(A_1^{[b]}, \ldots, A_n^{[b]})$ is a set of tuples $\langle c_1, \ldots, c_n \rangle$ such that each c_i is a constant belonging to abstract domain A_i. The binding pattern for a relation specifies which of the arguments of the relation must be bound by a constant in order to query the relation. For example, in relation r with schema $r(A_1, A_2, A_3^b)$, the first two arguments, corresponding resp. to abstract domains A_1 and A_2, are free, while the third argument, corresponding to A_3, is bound.

A *conjunctive query* (CQ) q of arity n over a set $\mathcal{R}$ of relational schemata is written in the form

$$q(X_1, \ldots, X_n) \;\leftarrow\; \mathit{conj}(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$$

where $q(X_1, \ldots, X_n)$ is called the *head* of q, and $\mathit{conj}(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ is called the *body* of q. The body of q is a conjunction of atoms whose predicate symbols are in $\mathcal{R}$, and that involve variables $X_1, \ldots, X_n, Y_1, \ldots, Y_m$, and possibly some constants. We assume that the query is *safe*, i.e., that each variable X_j appears in at least one atom in the body.

Given a database DB, the answer q^{DB} of q over DB is the set of tuples $\langle c_1, \ldots, c_n \rangle$ of constants in DB such that there are constants $d_1, \ldots, d_m$ for which each atom in $\mathit{conj}(c_1, \ldots, c_n, d_1, \ldots, d_m)$ holds in DB.

A *union of conjunctive queries* (UoCQ) q of arity n over a set $\mathcal{R}$ of relational schemata is a set $\{q_1, \ldots, q_k\}$ of CQs, each with head predicate q and arity n. Given a database DB, the answer q^{DB} of q over DB is the union of the answers to each conjunctive query q_i in q.

In the presence of access limitations on the relations, we say that a tuple is *obtainable* in the answer to a query if there exists a query plan that retrieves it. A *query plan* is a sequence of valid accesses to the relations. By *access* we mean a request, sent to a relation, imposing a single specific value for some (possibly no) argument in the relation. An access is *valid* if the request imposes a value for each bound arguments; the *response* to a valid access is the (possibly empty) set of tuples in the relation that matches the request. We observe that, in order to retrieve all obtainable tuples, relations may need to be accessed in a particular order, as shown in the following example, adapted from [5].

Example 1. Suppose we have two relations: $r_1(\mathit{Artist}^b, \mathit{Nation})$, which stores artists with their nationality, and $r_2(\mathit{Title}, \mathit{Year}^b, \mathit{Artist})$, which stores data

[1] Note that we use a positional notation for relations, and that the A_is do not denote attributes.

about songs. The query

$$q(T) \;\leftarrow\; r_1(A, \mathit{italian}),\; r_2(T, 1998, A)$$

asks for titles of songs interpreted by an Italian artist and produced in year 1998. Since r_1 requires the first position to be bound to a constant, accessing it first would not be valid. However, we could use artist names extracted from r_2 to access r_1 and extract tuples that may contribute to the answer. ■

In [7] an algorithm is presented that, given a query over the relations, retrieves all the obtainable tuples in the answer to the query. Such an algorithm consists in the evaluation of a suitable Datalog program that extracts all obtainable tuples starting from a set of initial values. The Datalog program is constructed by encoding in Datalog clauses the limitations on the relations that must be respected during query evaluation. It is assumed that the strategy of enumerating all possible elements of a given domain to access a relation is not feasible and that, rather, the values for the bound positions of a relation are obtained either from constants in the query or from tuples retrieved from other relations. The evaluation of the Datalog program makes use of a set storing the values extracted from the relations (together with the constants appearing in the query) at a certain step, and of a *cache* that stores a partial copy of the relations, populated with the tuples extracted from the relations at a certain step.

```
1. Initialize $B$ with the set of constants in the query
2. while there is a way of doing valid accesses with new constants
   (a) Access as many relations as possible, according to their binding patterns,
       using constants in $B$
   (b) Put the obtained tuples in the cache
   (c) Put the obtained constants in $B$
3. Evaluate the query over the cache
```

We call *binding values* the constants used to access a relation with access limitations; a *binding tuple* is a tuple, of the same arity as the number of bound arguments of a relation, whose values are binding values of the appropriate abstract domain that are used to access the same tuple. The algorithm extracts all tuples obtainable while respecting the binding patterns, performing a finite number of iterations of Step 2. Observe that there may be tuples in the relations that cannot be retrieved. We illustrate the extraction strategy by means of the following example.

Example 2. Consider the following set $\mathcal{R}$ of relational schemata

$$\begin{aligned}\mathcal{R} = \{\; & r_1(A^b, C),\\ & r_2(C, B^b),\\ & r_3(B, C^b)\;\;\}\end{aligned}$$

Suppose we have the following conjunctive query over $\mathcal{R}$:

$$q(B) \leftarrow r_1(a_1, C),\; r_2(C, B)$$

r_1 :

a_1	c_1
a_1	c_3
a_3	c_3

r_2 :

c_1	b_1
c_2	b_2
c_3	b_3

r_3 :

b_2	c_1
b_1	c_2

Fig. 1. Extension of relations of Example 2

Now, assume a database in which the relations have the extension shown in Figure 1. Starting from a_1, the only constant in the query, we access r_1 getting the tuples $\langle a_1, c_1\rangle$ and $\langle a_1, c_3\rangle$, whereas $\langle c_1, b_1\rangle$ in r_2 is not (yet) accessible because of the binding patterns. But now we have c_1 with which we can access r_3 and retrieve $\langle b_2, c_1\rangle$. With b_2 we extract $\langle c_2, b_2\rangle$ from r_2; then, with c_2 we retrieve $\langle b_1, c_2\rangle$ from r_3. Finally with b_1 we extract $\langle c_1, b_1\rangle$ from r_2 and obtain $\langle b_1\rangle$ as the answer to q. Observe that $\langle a_3, c_3\rangle$ and $\langle c_3, b_3\rangle$ could not be extracted from r_1 and r_2 respectively and that answer $\langle b_3\rangle$ is not obtainable. ∎

3 Query Planning for Conjunctive Queries

Given a query q over a set of relations $\mathcal{R}$, we want to construct a *query plan* that allows us to retrieve all obtainable answers to q. In this section we show how to generate an optimized query plan that avoids accesses that are not necessary for extracting all obtainable answers.

This problem has already been addressed for a subclass of UoCQs not covering CQs in [7, 8]. Building on a previous attempt to solve the problem for the full class of CQs [1], we improve on the techniques presented in [7] in two relevant directions.

– We consider queries that are in the full class of *unions of conjunctive queries.*
– We exploit the knowledge about the structure of the query to exclude accesses that are unnecessary to answer the query.

With regard to the second issue, we observe that, having extracted a number of values at a certain point of the query answering process, and given a relation r to be accessed using the values extracted so far as binding values, some of the possible accesses to r may not be necessary in order to calculate the answer to the query. This is illustrated in the following example.

Example 3. Consider the following set $\mathcal{R}$ of relational schemata

$$\begin{aligned}\mathcal{R} = \{\ & r_1(A^b, B),\\ & r_2(B^b, C),\\ & r_3(C^b, B)\ \ \}\end{aligned}$$

and the following CQ over $\mathcal{R}$:

$$q(C)\ \leftarrow\ r_1(a_0, B), r_2(B, C)$$

We observe that r_3 is not useful to answer the query. Using the values obtained from r_2 to access r_3 in order to obtain new values of domain B with which to access r_2 again is pointless. Indeed, to the join condition between r_1 and r_2 guarantees that the only tuples extracted from r_2 which can be used to construct a tuple of the answer to q are those obtained by binding the first argument of r_2 with a value extracted from r_1. ■

3.1 Queryable relations

We want to be able to restrict our attention to *queryable* relations, i.e., the relations that can be accessed at least once for at least one database instance, starting from the values in the query.

Definition 1. *Given a set $\mathcal{R}$ of relational schemata and a query q over $\mathcal{R}$, a relation r in $\mathcal{R}$ is said to be* queryable *if there exist a database DB such that, starting from the constants in the query, there is a sequence of valid accesses to the relations in $\mathcal{R}$ such that r is eventually accessed.*

Example 4. Let $\mathcal{R}$ be the following set of relational schemata:

$$\begin{aligned}\mathcal{R} = \{\ & r_1(A^b, B),\\ & r_2(B^b, C),\\ & r_3(B, B, C^b)\ \}\end{aligned}$$

Given the query

$$q() \ \leftarrow \ r_3(X, Y, c_1)$$

we show that r_2 and r_3 are queryable. To do so, we exhibit a database in which the extensions of the relations is as follows.

$$r_1 : \ \boxed{a_1 | b_1} \qquad r_2 : \ \boxed{b_1 | c_2} \qquad r_3 : \ \boxed{b_1 | b_2 | c_1}$$

Note that in the extraction process we start with the constant c_1 and we are able to access r_3, obtaining the tuple $\langle b_1, b_2, c_1 \rangle$; then, with the constants b_1 and b_2 we access r_2. This proves that r_3 and r_2 are queryable. Observe also that r_1 is *not* queryable: whatever are the tuples in r_2 and r_3, we cannot extract in any way from r_2 and r_3 values that belong to the abstract domain of A, which would be necessary to access r_1. ■

A correct and complete algorithm to calculate the queryable relations has been given in [7].

3.2 Dependency graphs and optimization thereof

We show now how to construct an optimized query plan for a CQ over some relations. We exclude a priori all non-queryable relations and therefore assume in the following that all the relations are queryable.

For our purposes, we define the *dependency graph* (or *d-graph*) of a CQ q with respect to a set $\mathcal{R}$ of relational schemata, denoted by $G_q^{\mathcal{R}}$. A d-graph is the structure on which we operate in order to eliminate unnecessary accesses to the sources; then, an "optimized" version of the d-graph will be used to generate the query plan for q.

We will construct the graph starting from a query that contains no constant. We therefore first illustrate a preprocessing step that eliminates constants from the query as follows. For each constant a appearing in q in the position of a domain A, we introduce a new relation r_a with a single free attribute, whose domain is that of A; the content of r_a is the single tuple $\langle a \rangle$. We then replace all occurrences of a appearing in the query in a position of domain A with a fresh new variable X_a, and we add the conjunct $r_a(X_a)$ to the body of q. The intuition is that a value acts as a relation whose content is completely known and accessible, and amounts only to the value itself.

The set $G_q^{\mathcal{R}}$ of nodes of the d-graph is determined as follows. For each atom in q, we have a set of nodes (called a *source*) in $G_q^{\mathcal{R}}$, one for each argument of the corresponding relation. We call such nodes *black*. Moreover, for each relation not appearing in q, we have a set of nodes (also called a *source*), one for each argument of the relation. We call such nodes *white*. Both types of nodes have two labels:

– The binding ("b" or "f") of the corresponding argument in the relation.
– The abstract domain of the corresponding argument in the relation.

As for the arcs, $G_q^{\mathcal{R}}$ has an arc from a node u to a node v whenever the following three conditions hold:

– u and v have the same abstract domain.
– u is free.
– v is bound.

Intuitively, the arcs denote dependencies between relation arguments, indicating that a relation with limited capabilities needs values which can be retrieved from other relations (or from constants in the query).

Our aim here is to optimize the set of arcs in $G_q^{\mathcal{R}}$ by removing dependencies that are actually not needed to retrieve all obtainable tuples.

For this purpose, let us indicate with $\texttt{outArcs}(u, G_q^{\mathcal{R}})$ the set of outgoing arcs from a node in the same source as node u, for a d-graph $G_q^{\mathcal{R}}$; we omit the second argument wherever the d-graph is understood. A sequence $u_1 \curvearrowright v_1, \ldots, u_n \curvearrowright v_n$ of arcs in $G_q^{\mathcal{R}}$ is called a *dependency path* (or *d-path*) for $G_q^{\mathcal{R}}$ if, for $2 \leq i \leq n$, $u_i \curvearrowright v_i \in \texttt{outArcs}(v_{i-1})$; to emphasize the first and last nodes in the sequence, the d-path can also be denoted $u_1 \curvearrowright^{+} v_n$. Then, we can eliminate the arcs for which no d-path exists that reaches a black node, i.e., we must avoid accesses to relations that cannot provide values that can contribute to the answer.

Moreover, we may eliminate some arcs thanks to the presence of joins in the query. We say that an arc $u \curvearrowright v$ is *strong* whenever (i) both u and v are black, (ii) u and v correspond to two variables which are joined in the query, and (*iii*) v's source is not needed to provide arbitrary values to other relations used in the query; all other arcs are called *weak* arcs. Now, in the presence of a strong arc, the join indicates that *all* the useful tuples that can be retrieved from v's relation are extracted using *only* values coming from u. Therefore, whenever a node has an incoming strong arc, all the incoming weak arcs must be deleted, provided that this does not affect queryability of the relation as made precise below. We say that such weak arcs are *dominated* by the strong arc(s).

We say that a relation is *free* if all of its attributes are free; a source is *free* if all of its nodes correspond to free attributes. A bound node v in a d-graph is inductively defined as *free-reachable*, denoted as $\mathit{isFreeReachable}(v)$, if either (*i*) there is a weak arc $u \frown v$ such that all bound nodes in u's source are free-reachable or (*ii*) all strong arcs $u_1 \frown v, \ldots, u_n \frown v$ are such that all bound nodes in u_i's source are free-reachable, for $1 \leq i \leq n$. Clearly, whenever the query is constant-free (like after the pre-processing step), a relation is queryable only if all of its bound nodes are free-reachable.

Let *isBlack* be a function that takes a node and returns *true* if and only if the node is black, and let *joined* be a function that takes two nodes and returns *true* if and only if the corresponding variables are joined in the query. Based on the above observations, we characterize strong arcs and deleted arcs by the following equations (1) and (2).

$$\begin{aligned} \mathit{isStrong}(u \frown v) = {} & \neg \mathit{isDeleted}(u \frown v) \wedge \\ & \mathit{isBlack}(u) \wedge \mathit{isBlack}(v) \wedge \mathit{joined}(u, v) \wedge \\ & \forall \gamma \in \mathtt{outArcs}(v)(\mathit{isStrong}(\gamma) \vee \mathit{isDeleted}(\gamma)) \wedge \\ & \mathit{isFreeReachable}(v) \end{aligned} \tag{1}$$

$$\begin{aligned} \mathit{isDeleted}(u \frown v) = {} & \neg \mathit{isStrong}(u \frown v) \wedge \\ & [\mathit{isBlack}(v) \wedge \exists (u' \neq u)(\mathit{isStrong}(u' \frown v)) \vee \\ & \neg \mathit{isBlack}(v) \wedge \forall \gamma \in \mathtt{outArcs}(v)(\mathit{isDeleted}(\gamma))] \end{aligned} \tag{2}$$

Ideally, for a given d-graph, we aim to determine two maximal sets respectively of deleted arcs and strong arcs that satisfy the above equations. Let us call *candidate strong* arc any arc whose nodes are black and whose corresponding variables are joined in the query; let us indicate with $\mathtt{arcs}(G_q^{\mathcal{R}})$ the set of all arcs in $G_q^{\mathcal{R}}$ and with $\mathtt{cand}(G_q^{\mathcal{R}})$ the set of all candidate strong arcs in $G_q^{\mathcal{R}}$. Clearly, (*i*) only candidate strong arcs have the potential to become strong arcs and (*ii*) no candidate strong arc can ever be deleted, since, by definition, it reaches a black node and is never dominated by another strong arc (for any given node in $G_q^{\mathcal{R}}$, its incoming arcs, if any, are either all strong or all weak). Therefore the set of strong arcs and the set of deleted arcs must be disjoint; this is reflected in the first conjunct of equation (1) and equation (2). Note also that only those candidate strong arcs that do not destroy free-reachability can actually become strong. In particular, we say that a candidate strong arc $u \frown v$ is *circular* if it is contained in a d-path $u \frown^{+} u$ such that all arcs in it are candidate strong; we indicate by $\mathtt{circ}(G_q^{\mathcal{R}})$ the set of circular candidate strong arcs in $G_q^{\mathcal{R}}$.

We call the pair $(\mathcal{S}, \mathcal{D})$ a *solution* for equations (1) and (2) if $\mathcal{S}$ and $\mathcal{D}$ are respectively sets of strong arcs and deleted arcs (among the arcs of a given dependency graph) that satisfy the conditions of the equations; the solution is *maximal* if no other solution $(\mathcal{S}', \mathcal{D}')$ exists such that $\mathcal{S}' \supset \mathcal{S}$ or $\mathcal{D}' \supset \mathcal{D}$. A solution $(\mathcal{S}, \mathcal{D})$ can be used to produce a new d-graph $G_q^{\mathcal{R}(\mathcal{S},\mathcal{D})}$, that we call *optimized d-graph*, by removing from $G_q^{\mathcal{R}}$ all arcs in $\mathcal{D}$ and by labeling as "strong" all arcs in $\mathcal{S}$, by labeling as "weak" all remaining arcs (note that the input d-graph $G_q^{\mathcal{R}}$ had no explicit arc labeling), and, finally, by removing all nodes with no incoming or outgoing arcs and all sources with no nodes. It turns out that there always exists a unique maximal solution for equations (1) and (2).

Lemma 1. *Let $(\mathcal{S}, \mathcal{D})$ be a solution for (1') and (2) for a d-graph G, where (1') is as (1), but without the last conjunct. A bound node u is free-reachable in $G^{(\mathcal{S},\mathcal{D})}$ iff there is no d-path $u \frown^{+} u$ in $G^{(\mathcal{S},\mathcal{D})}$ consisting only of strong arcs.*

Theorem 1. *Equations (1) and (2) admit a unique maximal solution for any d-graph G.*

Proof. We first prove that a solution exists. Consider equations (3) and (4).

$$\begin{aligned} \mathit{isStrong}(u \frown v) \equiv\ & \mathit{isBlack}(u) \wedge \mathit{isBlack}(v) \wedge \mathit{joined}(u, v) \wedge \\ & \forall \gamma \in \mathtt{outArcs}(v)(\mathit{isStrong}(\gamma) \vee \mathit{isDeleted}(\gamma)) \end{aligned} \tag{3}$$

$$\begin{aligned} \mathit{isDeleted}(u \frown v) \equiv\ & [\mathit{isBlack}(v) \wedge \exists u'(\mathit{isStrong}(u' \frown v)) \vee \\ & \neg \mathit{isBlack}(v) \wedge \forall \gamma \in \mathtt{outArcs}(v)(\mathit{isDeleted}(\gamma))] \end{aligned} \tag{4}$$

They induce the definition of two corresponding monotonic fixpoint operators T_S and T_D. This in turn guarantees the existence of the greatest fixpoint for (3) and (4) (the one that maximizes the number of strong arcs and deleted arcs), for any given initial sets of strong arcs and of deleted arcs.

By Lemma 1, the largest set of strong arcs one can hope for is $\mathcal{S}^0 = \mathtt{cand}(G) \setminus \mathtt{circ}(G)$, whereas the largest achievable set of deleted arcs is its complement $\mathcal{D}^0 = \mathtt{arcs}(G) \setminus \mathcal{S}^0$; clearly, $\mathcal{S}^0$ and $\mathcal{D}^0$ are disjoint. It now suffices to observe that the fixpoint $(\mathcal{S}^\omega, \mathcal{D}^\omega)$ for equations (3) and (4) obtained via $T_S \circ T_D$ starting from $(\mathcal{S}^0, \mathcal{D}^0)$ is necessarily also a solution for equations (1) and (2), since, by monotonicity, $\mathcal{S}^0 \cap \mathcal{D}^0 = \emptyset$ entails that $\mathcal{S}^\omega \cap \mathcal{D}^\omega = \emptyset$, so all conditions in (1) and (2) are satisfied, including free-reachability by Lemma 1.

To see that the maximal solution for (1) and (2) is unique and coincides with $(\mathcal{S}^\omega, \mathcal{D}^\omega)$, let us assume, by contradiction, that there is another solution $(\mathcal{S}', \mathcal{D}')$ such that $\mathcal{S}' \supset \mathcal{S}^\omega$ or $\mathcal{D}' \supset \mathcal{D}^\omega$. However, since $(\mathcal{S}', \mathcal{D}')$ is a solution, we have $\mathcal{S}' \cap \mathcal{D}' = \emptyset$. Besides, $\mathcal{S}' \subseteq \mathcal{S}^0$, since only non-circular candidate strong arcs can be strong arcs, and $\mathcal{D}' \subseteq \mathcal{D}^0$, since no candidate strong arc can be deleted, as was argued before. This means that $(\mathcal{S}', \mathcal{D}')$ would also be a solution for equations (3) and (4), contradicting the fact that (3) and (4) are guaranteed to have a unique greatest fixpoint. □

Theorem 1 indicates that we can always find the largest sets of strong arcs and of deleted arcs possible for a given dependency graph. We present in Figure 2 an algorithm that determines such sets according to the above criteria. The algorithm calculates the initial sets of strong arcs (the non-circular candidate strong arcs) and deleted arcs (its complement) and then implements the T_S and T_D monotonic fixpoint operators used in the proof of Theorem 1 and applies them to the sets until a fixpoint is reached. This is stated below in Theorem 2, which trivially holds.

Theorem 2. *The function calculateGFP(G) shown in Figure 2 computes the maximal solution for equations (1) and (2) with respect to d-graph G.*

Termination of the algorithm is guaranteed by the monotonicity of the fixpoint operators induced by equations (3) and (4), which are implemented by the functions *unmarkStrong* and *unmarkDeleted* respectively. The monotonicity also ensures that the algorithm runs in polynomial time in the size of the d-graph.

Theorem 3. *calculateGFP(G) runs in polynomial time in the size of G.*

An optimized d-graph generated via the result of the function *calculateGFP* trivially has the properties stated in Proposition 1 below.

Proposition 1. *Consider an optimized d-graph $G_q^{\mathcal{R}}$ for a query q over a set $\mathcal{R}$ of relational schemata. We have that $G_q^{\mathcal{R}}$ has the following properties.*

1. *From each node in $G_q^{\mathcal{R}}$ it is possible to reach a black node through a d-path.*
2. *For each node u, its incoming arcs, if any, are either all strong or all weak.*
3. *There is no d-path that traverses a strong arc and successively a weak arc.*
4. *From every node of $G_q^{\mathcal{R}}$, a free source is reachable through a d-path in reverse direction.*

Example 5. Let $\mathcal{R} = \{r_1(A^b, B), r_2(A, B^b)\}$ be a set of relational schemata and $q(X) \leftarrow r_1(a, X)$ a query over $\mathcal{R}$. The d-graph $G_q^{\mathcal{R}}$ for q is shown in Figure 3, where we have named the sources as the corresponding relations, with a superscript indicating the occurrence number of that relation in the query. We first eliminate the constant a occurring in q by introducing a new relation r_a with domain A and populated by the single tuple $\langle a \rangle$ and by rewriting q as follows:

$$q(X) \leftarrow r_a(Y), r_1(Y, X).$$

Arc e_1 is the only (non-circular) candidate strong arc and is, thus, in the initial set of strong arcs; arcs e_2 and e_3 are therefore in the initial set of deleted arcs. This is already the greatest fixpoint we were looking for, since equations (3) and (4) are satisfied. In particular, arc e_3 remains deleted, since it is dominated by e_1, which is strong, and then e_2 is deleted as well, since no black node is reachable by a d-path starting with e_2. The intuition is that relation r_1 does not have to provide arbitrary values to r_2; indeed, due to the join condition in q, accessing r_1 with values provided by r_2 would not provide tuples that could be used to answer the query q. The optimized d-graph, without deleted arcs, and without source r_2, is shown in Figure 4; the strong arc e_1 is denoted by a solid line. ∎

```
$\textit{calculateGFP}(G$ : d-graph$)$ : arc set $\times$ arc set
  $\mathcal{S} := \texttt{cand}(G) \setminus \texttt{circ}(G)$
  $\mathcal{D} := \texttt{arcs}(G) \setminus \mathcal{S}$
  do {
    $(\mathcal{S}',\mathcal{D}') := (\mathcal{S},\mathcal{D})$
    $\mathcal{S} :=$ unmarkStrong$(\mathcal{S}',\mathcal{D}',G)$
    $\mathcal{D} :=$ unmarkDeleted$(\mathcal{S}',\mathcal{D}',G)$
  } while $(\mathcal{S},\mathcal{D}) \neq (\mathcal{S}',\mathcal{D}')$
  return $(\mathcal{S}',\mathcal{D}')$

$\textit{unmarkStrong}(\mathcal{S}$ : arc set, $\mathcal{D}$ : arc set, $G$ : d-graph$)$ : arc set
  $\mathcal{S}' := \mathcal{S}$
  for each arc $u \frown v \in \mathcal{S}$
    for each arc $\gamma \in \texttt{outArcs}(v, G)$
      if $(\gamma \notin \mathcal{S} \cup \mathcal{D})$ $\mathcal{S}' := \mathcal{S}' \setminus \{u \frown v\}$; break
  return $\mathcal{S}'$

$\textit{unmarkDeleted}(\mathcal{S}$ : arc set, $\mathcal{D}$ : arc set, $G$ : d-graph$)$ : arc set
  $\mathcal{D}' := \mathcal{D}$
  for each arc $u \frown v \in \mathcal{D}$
    if $(\textit{isBlack}(v))$ then
      bool strongExists := false
      for each arc $u' \frown v' \in \mathcal{S}$
        if $(v = v')$ then strongExists := true ; break
      if (not strongExists) then $\mathcal{D}' := \mathcal{D}' \setminus \{u \frown v\}$
    else ($v$ is white)
      if $(\texttt{outArcs}(v, G) \setminus \mathcal{D} \neq \emptyset)$ then $\mathcal{D}' := \mathcal{D}' \setminus \{u \frown v\}$
  return $\mathcal{D}'$
```

Fig. 2. Algorithm determining the maximal sets of strong arcs and deleted arcs

3.3 Generation of a query plan

Now we come to the construction of the query plan. From the resulting optimized d-graph we construct the optimized query plan, expressed in Datalog notation, as shown in the algorithm of Figure 5. The Datalog program is evaluated according to the usual least fixpoint semantics; it is guaranteed by construction that the least fixpoint can be calculated by only making valid accesses.

The algorithm rewrites the original query over new versions of the relations in the body. For each predicate r in the body of the query, we introduce a new predicate with the same arity as r that acts as a sort of *cache* in which we store, during the query answering process, all the tuples extracted from r. This is done in step 1.2 of the algorithm of Figure 5. Note that different occurrences of the same predicate give rise to different names; in the examples we choose, e.g., to add a hat symbol to the predicate name as well as an occurrence number.

Each cache relation is defined in step 2 of the algorithm as the corresponding original relation, but where each bound variable receives its binding values from another new relation created for that purpose in step 2.3. Such relation takes

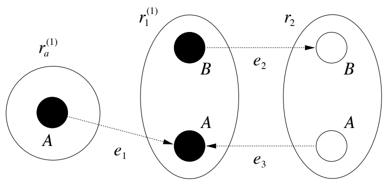

Fig. 3. Dependency graph for Example 5

into account the arcs in the d-graph: if the corresponding incoming arcs are weak (resp., strong), then in step 2.3.2 (resp., step 2.3.3) the relation is defined as a disjunction (resp., conjunction) of the cache relations corresponding to the origin nodes, since any of them (resp., only their join) can provide binding values.

Finally, the program generated by the algorithm of Figure 5 is completed by adding a fact for each relation created in the preprocessing step to eliminate the constants from the query; the fact has the form $r_a(a)$, where r_a is the created relation and a is the removed constant.

Example 6. We refer to Example 5. From the optimized dependency graph relative to q, the algorithm of Figure 5 generates the following plan:

$$\begin{aligned} q(X) &\leftarrow \hat{r}_a^1(X), \hat{r}_1^1(X,Y) \\ \hat{r}_a^1(X_1) &\leftarrow r_a(X_1) \\ \hat{r}_1^1(X_2,Y_2) &\leftarrow r_1(X_2,Y_2), s^{X_2}(X_2) \\ s^{X_2}(X_2) &\leftarrow \hat{r}_a^1(X_2) \\ r_a(a) &\leftarrow \end{aligned}$$

In this example, the support relation s^{X_2} introduced by step 2.3.3 is defined as r_a^1, since there is only one corresponding incoming strong arc. The third and fourth rules could therefore more simply be written as the following single rule:

$$\hat{r}_1^1(X_2,Y_2) \leftarrow r_1(X_2,Y_2), \hat{r}_a^1(X_2)$$

The Datalog program above shows that relation r_2, which could in principle provide useful values to r_1, is not needed at all to answer the query because of the join between r_a and r_1. ∎

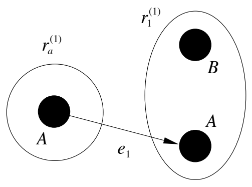

Fig. 4. optimized d-graph for Example 5

```
INPUT: optimized d-graph $G_q^{\mathcal{R}}$, query $q$, relations $\mathcal{R}$
OUTPUT: a Datalog program corresponding to the optimized query plan

1 Rewrite $q$ as follows:
  1.1 The query head is the same
  1.2 Each atom in $q$'s body is replaced by an atom with the same arguments but
      with a fresh new relation name, henceforth used to uniquely identify the cor-
      responding source
2 For each atom $p$ of arity $n$ in the body of the rewritten query $q$ or in the schema of
  a relation not in $q$ with incoming or outgoing arcs, a rule is generated as follows:
  2.1 The head atom is $p(Y_1, \ldots, Y_n)$, where the $Y_i$'s are fresh new variables
  2.2 The body has one atom $r(Y_1, \ldots, Y_n)$, where $r$ is the name of the relation
      corresponding to $p$'s source
  2.3 The body also has one atom for each bound node $v$ in $q$'s head with incoming
      arcs:
    2.3.1 The atom has a fresh new relation name that uniquely identifies $v$ and
          only 1 argument containing the variable corresponding to $v$ in the head
    2.3.2 If $v$'s incoming arcs are weak, create one new rule for each arc $u^\frown v$ as
          follows:
      2.3.2.1 The head is the atom created in step 2.3.1
      2.3.2.2 The body has one atom. The relation name is $u$'s source identifier.
              All positions in the atom have a new variable, except for the position
              corresponding to $u$, which has the same variable as the one in the
              head
    2.3.3 If $v$'s incoming arcs are strong, create one new rule.
      2.3.3.1 The head is the atom created in step 2.3.1
      2.3.3.2 The body has one atom for each arc $u^\frown v$. The relation name is $u$'s
              source identifier. All positions in the atom have a new variable, except
              for the position corresponding to $u$, which has the same variable as
              the one in the head
```

Fig. 5. Algorithm producing the Datalog program corresponding to the optimized query plan

Such a Datalog program ensures that the binding values for the bound positions of a relation r are obtained from values retrieved from the appropriate relations and stored in the caches.

We now state that the optimized query plan generated by our approach is both sound, i.e., it returns only tuples that are in the answer to the query, and complete, i.e., it does not miss any obtainable tuple, taking into account the binding patterns.

Theorem 4. *Let q be a query over a set $\mathcal{R}$ of relational schemata, and let $G_q^{\mathcal{R}}$ be its d-graph and $G = calculateGFP(G_q^{\mathcal{R}})$ the corresponding optimized d-graph. Then, the Datalog program constructed from q by the algorithm in Figure 5, based* *on G, computes all the answers to q obtainable from the relations in $\mathcal{R}$, given the binding patterns on them.*

4 Extension to unions of conjunctive queries

In this section, we address the problem of finding a query plan for a UoCQ. Our approach is to transform a UoCQ into an appropriate CQ, so that the technique presented in Section 3 can be applied; the output is then adjusted so as to obtain a query plan for the original UoCQ (assumed, w.l.o.g., to be constant-free).

Let q be a constant-free UoCQ over a set $\mathcal{R}$ of relational schemata defined as follows:

$$\begin{array}{rcl} \{ \quad q(X_{1,1},\ldots,X_{n,1}) & \leftarrow & \mathit{conj}_1(X_{1,1},\ldots,X_{n,1},Y_{1,1},\ldots,Y_{k,1}), \\ & \vdots & \\ q(X_{1,m},\ldots,X_{n,m}) & \leftarrow & \mathit{conj}_m(X_{1,m},\ldots,X_{n,m},Y_{1,m},\ldots,Y_{k,m}) \ \} \end{array}$$

We first replace each rule in q with a variant thereof so that in the end all rules are standardized apart, as follows

$$\begin{array}{rcl} \{ \quad q(X'_{1,1},\ldots,X'_{n,1}) & \leftarrow & \mathit{conj}_1(X'_{1,1},\ldots,X'_{n,1},Y'_{1,1},\ldots,Y'_{k,1}), \\ & \vdots & \\ q(X'_{1,m},\ldots,X'_{n,m}) & \leftarrow & \mathit{conj}_m(X'_{1,m},\ldots,X'_{n,m},Y'_{1,m},\ldots,Y'_{k,m}) \ \} \end{array}$$

where each primed variable is a fresh new variable. Clearly, the expression above is equivalent to the original UoCQ.

We then generate the following CQ:

$$\begin{array}{rcl} q'(X'_{1,1},...,X'_{n,1},...,X'_{1,m},...,X'_{n,m}) & \leftarrow & \mathit{conj}_1(X'_{1,1},...,X'_{n,1},Y'_{1,1},...,Y'_{k,1}), \\ & & \ldots, \\ & & \mathit{conj}_m(X'_{1,m},...,X'_{n,m},Y'_{1,m},...,Y'_{k,m}) \end{array}$$

Now we apply to q' the technique developed for CQs in Section 3. We obtain as output a Datalog program Π whose first clause redefines q' and has the following form:

$$\begin{array}{rcl} q'(X'_{1,1},...,X'_{n,1},...,X'_{1,m},...,X'_{n,m}) & \leftarrow & \hat{\mathit{conj}}_1(X'_{1,1},...,X'_{n,1},Y'_{1,1},...,Y'_{k,1}), \\ & & \ldots, \\ & & \hat{\mathit{conj}}_m(X'_{1,m},...,X'_{n,m},Y'_{1,m},...,Y'_{k,m}) \end{array}$$

where the various $\hat{\mathit{conj}}_i$'s are as the conj_i's, but refer to the cache relations instead, which are defined in the remainder of Π, which we denote as Π'.

Finally, from the rewritten query q', we generate the following m rules.

$$\begin{array}{rcl} q(X'_{1,1},...,X'_{n,1},...,X'_{1,m},...,X'_{n,m}) & \leftarrow & \hat{\mathit{conj}}_1(X'_{1,1},...,X'_{n,1},Y'_{1,1},...,Y'_{k,1}) \\ & & \ldots, \\ q(X'_{1,1},...,X'_{n,1},...,X'_{1,m},...,X'_{n,m}) & \leftarrow & \hat{\mathit{conj}}_m(X'_{1,m},...,X'_{n,m},Y'_{1,m},...,Y'_{k,m}) \end{array}$$

The above m rules together with Π' constitute a Datalog program corresponding to an optimized query plan that computes all the answers to the original UoCQ q that are obtainable from the relations in $\mathcal{R}$.

5 Discussion

We have presented a technique that can be used to find all obtainable tuples belonging to the answer to a query formulated over relations with access limitations. Based on the binding patterns of a query, we construct a d-graph that indicates all possible ways in which bound arguments in a relation can receive useful values from free arguments with the same domain. The d-graph is optimized according to the joins included in the query and then used to generate a Datalog program that answers the query. The technique works for CQs as well as UoCQs.

Currently, we are working on an extension of our method to guarantee minimality of accesses; this requires considering the structure of the query in its entirety, including atoms that have no variable in common with the rest of the query. We are also aiming to cover full Datalog queries. The idea is that, before considering weak and strong arcs in the d-graph, one should take into account "unfolding" arcs, i.e., arcs that relate the arguments of an occurrence of an intensional predicate with the arguments of the predicates defining it.

References

1. Andrea Calì and Diego Calvanese. Optimized querying of integrated data over the Web. In *Proc. of the IFIP WG8.1 Working Conference on Engineering Information Systems in the Internet Context (EISIC 2002)*, pages 285–301. Kluwer Academic Publishers, 2002.
2. Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *Proc. of IJCAI'97*, pages 778–784, 1997.
3. Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.
4. Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *Proc. of ACM SIGMOD*, pages 311–322, 1999.
5. Alon Y. Levy. Answering queries using views: A survey. Technical report, University of Washinghton, 1999.
6. Chen Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB Journal*, 12(3):211–227, 2003.
7. Chen Li and Edward Chang. Query planning with limited source capabilities. In *Proc. of ICDE 2000*, pages 401–412, 2000.
8. Chen Li and Edward Chang. Answering queries with useful bindings. *ACM Trans. on Database Systems*, 26(3):313–343, 2001.
9. Chen Li and Edward Chang. On answering queries in the presence of limited access patterns. In *Proc. of ICDT 2001*, pages 219–233, 2001.
10. Todd D. Millstein, Alon Y. Levy, and Marc Friedman. Query containment for data integration systems. In *Proc. of PODS 2000*, pages 67–75, 2000.
11. Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proc. of PODS'95*, 1995.