

Querying Data under Access Limitations

Andrea Cali^{1,3}, Davide Martinenghi^{2,3}

¹*Computing Laboratory and Oxford-Man Institute of Quantitative Finance, The University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom
andrea.cali@comlab.ox.ac.uk*

²*Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
martinen@elet.polimi.it*

³*Faculty of Computer Science, Free University of Bozen-Bolzano
Piazza Domenicani 3, 39100 Bolzano, Italy*

Abstract—Data sources on the web are often accessible through web interfaces that present them as relational tables, but require certain attributes to be mandatorily selected, e.g., via a web form. In a scenario where we integrate a set of such sources, and we pose queries over them, the values needed to access a source may have to be retrieved from other sources that are possibly not even mentioned in the query: answering queries at best can then be done only with a potentially recursive query plan that gets all obtainable answers to the query. Since data sources are typically distributed over a network, a major cost indicator for the execution of a query plan is the number of accesses to remote sources. In this paper we present an optimization technique for conjunctive queries that produces a query plan that: (1) minimizes the number of accesses according to a strong notion of minimality; (2) excludes all sources that are not relevant for the query. We introduce *Toorjah*, a prototype system that answers queries posed on sources with limitations by means of optimized query plans. *Toorjah* adopts a strategy that is aimed to retrieve answers as early as possible during query processing, and to present them to the user as they are computed. We provide experimental evidence of the effectiveness of our optimization, by showing the reduction of the number of accesses in a large number of cases.

I. INTRODUCTION

In the context of integration of web data [1], or in a data exchange setting where source data are retrieved on the web, information is often accessible only via forms; it is easy to see that accessing data through a web form amounts to querying a relational table, where a selection is specified by the fields that are filled in. Typically, certain fields are required to be filled in by the user in order to obtain a result; for example, all online shops forbid a request posed by a user who leaves all fields of the search form empty. Analogously, in legacy systems where data are scattered over several files and wrapped as relational tables, similar limitations are enforced.

Limitations on how sources can be accessed significantly complicate query processing: as shown, e.g., in [2], [3], query answering in the presence of access limitations in general requires the evaluation of a *recursive* query plan. This is shown in the following example.

Example 1: Suppose we have three relations:
 $r_1(\text{Artist}, \text{Nation}, \text{YOB})$, which stores artists with their nationality and year of birth, and requires the first attribute to be selected; $r_2(\text{Title}, \text{Year}, \text{Artist})$, which stores data

about songs, and requires the second attribute to be selected; $r_3(\text{Artist}, \text{Album})$, that stores artists and albums of which they are authors, and has no fill-in attributes. The query $q(N) \leftarrow r_1(A, N, Y_1), r_2(\text{'volare'}, Y_2, A)$ asks for the nationality of the artist(s) who wrote 'volare'. Since there are no selections on the fill-in attributes, there is no way of answering this query in the “traditional” way. However, the best we can do to find answers to q is accessing first r_3 , and with all the retrieved artist names then access r_1 ; the returned tuples (if any) will have some constant for the attribute *YOB*, that we can use to access r_2 ; furthermore, with the artist names in the tuples extracted from r_2 we can access r_1 , and so on, until no more tuples can be extracted. Finally, after this possibly lengthy process, calculating the join specified by q on the extracted tuples will return all obtainable answers to q under the given access limitations. Notice that this requires the access to a relation (r_3) that is not even mentioned in the query; also, we need to consider the fact that both attributes *YOB* and *Year* represent values of the same kind – this will be formally represented by the notion of *abstract domain*. ■

As we can see in the previous example, returning the largest set of answers to a query under access limitations may require the evaluation of a recursive query plan, since values extracted from a source can be used to access another source. Since accesses to web and legacy sources are usually slow, a fundamental issue is how to reduce the *number* of accesses to the sources while still returning all obtainable answers to a query.

In this paper we address the problem of query plan optimization under access limitations. We present a graph-based representation of conjunctive queries and show how to prune the graph in order to determine which sources are relevant to a given query. We then show how to generate a minimal query plan, according to a strong notion of minimality, by making use of the information about the limitations and the structure of the query. We discuss an execution strategy for the query plan and present *Toorjah*¹, a prototype system that performs

¹Tool for Optimizing On-the-web Requests with Joins under Access Hindrances. *Toorjah* is also a phonetic rewriting of “turgia”, that in Piedmont, Italy, denotes an infertile cow, from which a tasty salami is produced.

query answering by using minimal query plans; also, Toorjah tries to access data sources in parallel, so as to obtain answers early in the query answering process, and to present them to the user as soon as they are retrieved. We give experimental evidence of the effectiveness of our optimization techniques, showing how a minimal plan avoids unnecessary accesses.

II. PRELIMINARIES

We consider relations as sets of tuples of values belonging to given domains and accessible via given access 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 song titles. A *relation schema* is a signature of the form $r^\alpha(A_1, \dots, A_n)$, where r is the relation name, n is called the *arity* of the relation, each A_i is an abstract domain², and α is an *access pattern* for r , i.e., a sequence of ‘*i*’ and ‘*o*’ symbols of length n ; for $1 \leq k \leq n$, the k -th argument of r is said to be an *input* argument if the k -th symbol in α is ‘*i*’, an *output* argument otherwise. If the access pattern does not contain any ‘*i*’ then the relation is said to be *free*. A *database schema* (or, simply, schema) is a set of relation schemata for different relations. A *relation* over a relation schema $r^\alpha(A_1, \dots, A_n)$ is a set of tuples $\langle c_1, \dots, c_n \rangle$ such that each c_i is a value belonging to abstract domain A_i . The input arguments indicated in the access pattern are those that must be bound by a value in order to query the relation. For example, in relation r with schema $r^{ooi}(A_1, A_2, A_3)$, the first two arguments, corresponding respectively to abstract domains A_1 and A_2 , are output arguments, while the third argument, corresponding to A_3 , is an input argument and thus has to be selected with a value for r to be queried. A (*database*) *instance* of a schema \mathcal{R} is a set of relations, one over each relation schema in \mathcal{R} . For convenience of notation, we sometimes indicate a sequence of terms t_1, \dots, t_n as \vec{t} and a tuple $\langle c_1, \dots, c_m \rangle$ as $\langle \vec{c} \rangle$; the length of a sequence \vec{t} is denoted by $|\vec{t}|$. A *conjunctive query* (CQ) q of arity n over a schema \mathcal{R} is written in the form $q(\vec{X}) \leftarrow \text{conj}(\vec{X}, \vec{Y})$, where $|\vec{X}| = n$, $q(\vec{X})$ is called the *head* of q , $\text{conj}(\vec{X}, \vec{Y})$ is called the *body* of q and is a conjunction of atoms involving the variables in \vec{X} and \vec{Y} and possibly some constants, and the predicate symbols of the atoms are in \mathcal{R} .

Given a database D over a schema \mathcal{R} , the *answer* q^D to a CQ q over \mathcal{R} is the set of tuples $\langle \vec{c} \rangle$ of constants, with $|\vec{c}| = |\vec{X}|$, such that there is a sequence of constants \vec{d} , with $|\vec{d}| = |\vec{Y}|$, for which each atom in $\text{conj}(\vec{c}, \vec{d})$ holds in D .

A *union of conjunctive queries* (UCQ) q of arity n over a schema \mathcal{R} is a set $\{q_1, \dots, q_k\}$ of CQs, each with head predicate q and arity n . Given a database D , the answer q^D to q over D is the union of the answers to each CQ in q .

In the following, we shall make use of the notion of *access*: informally, an access is the smallest operation that can be

²Note that we use a positional notation for relations, and that the A_i ’s do not denote attributes.

- 1) Initialize B with the set of constants in the query
- 2) **while** accesses can be made with new values
 - a) Access all possible relations, according to their access patterns, using values in B
 - b) Put the obtained tuples in the cache
 - c) Put the obtained constants in B
- 3) Evaluate the query over the cache

Fig. 1. Naive approach to query evaluation

performed on relations with access limitations, and it consists of the evaluation of a CQ with one body atom over a relation, where all the input attributes are selected with constants, and all output attributes are non-selected.

A relation can be accessed either if it has no input arguments or if some constants that can bind its input arguments are known. Then, as soon as new constants are extracted with an access, these can be used to make more accesses. Such sequences of accesses constitutes an *access plan*. For each database instance, each query should then be associated with an access plan that extracts the answers to the query. This is precisely what a *query plan* does.

In [3] an algorithm (sketched in Figure 1) is presented that, given a query over a schema, retrieves all the obtainable tuples in the answer to the query. Such an algorithm consists in the evaluation of a suitable Datalog program, encoding the access limitations in the schema, that extracts all obtainable tuples starting from a set of initial values. 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 input positions of a relation are obtained either from constants in the query or from tuples retrieved from other relations. The evaluation makes use of a set storing the values as they are extracted from the relations (together with the constants appearing in the query), and of a *cache* populated with the tuples retrieved so far. The algorithm extracts all tuples obtainable while respecting the access patterns. Observe that there may be tuples in the relations that cannot be retrieved. Also, access constraints on the schema may prevent some relations from being accessed, independently of the database instance. We want to be able to disregard such relations and to restrict our attention to *queryable* relations, i.e., those relations that can be accessed at least once for at least one database instance, starting from the values in the query. A query is *answerable* if and only if no non-queryable relation occurs in it.

Example 2: Consider the schema $\mathcal{R} = \{r_1^{io}(A, C), r_2^{io}(B, C), r_3^{io}(C, B)\}$ and the CQ $q_1(B) \leftarrow r_1(a_1, C), r_2(B, C)$ over \mathcal{R} . Assume that the relations have the extensions $r_1 = \{\langle a_1, c_1 \rangle, \langle a_1, c_3 \rangle\}$, $r_2 = \{\langle b_1, c_1 \rangle, \langle b_2, c_2 \rangle, \langle b_3, c_3 \rangle\}$, $r_3 = \{\langle c_1, b_2 \rangle, \langle c_2, b_1 \rangle\}$. Starting from a_1 , the only constant in the query, we access r_1 getting the tuples $\langle a_1, c_1 \rangle$ and $\langle a_1, c_3 \rangle$. Since we have c_1 , we can access r_3 and retrieve $\langle c_1, b_2 \rangle$. With b_2 we extract $\langle b_2, c_2 \rangle$ from r_2 ; then, with c_2 we retrieve $\langle c_2, b_1 \rangle$ from r_3 . Finally with b_1 we extract $\langle b_1, c_1 \rangle$ from r_2 and obtain $\langle b_1 \rangle$ as the answer to q_1 . Observe that $\langle b_3, c_3 \rangle$ could not be extracted from r_2 and, therefore, that answer $\langle b_3 \rangle$ is not obtainable.

Consider now the query $q_2(X) \leftarrow r_3(X, c_1)$. To show that r_2 and r_3 are queryable, we exhibit a database in which the extensions of the relations are as follows: $r_1 = \{\langle a_1, c_1 \rangle\}$, $r_2 = \{\langle b_2, c_2 \rangle\}$, $r_3 = \{\langle c_1, b_1 \rangle\}$. Starting with the constant c_1 , we are able to access r_3 , obtaining the tuple $\langle b_1, c_1 \rangle$; then, with the constant b_1 we access r_2 . This proves that r_3 and r_2 are queryable wrt. q_2 . Instead, r_1 is *not* queryable wrt. q_2 : independently of 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 are necessary to access r_1 . Query q_2 is answerable since r_3 is queryable. ■

An algorithm to calculate the queryable relations in a schema has been given in [3].

III. DEPENDENCY GRAPHS

Although we have shown in Section II how to retrieve all obtainable answers to a query over a schema with access limitations, the access plans executed by such algorithm make, in general, several accesses that turn out to be unneeded. Indeed, not all the relations of a schema may be necessary to access other relations and, eventually, to obtain the answer to the query; in particular, some of the relations that are not mentioned in the query may be always useless. This kind of usefulness is captured by the notion of *relevance* [4], which is illustrated in the next example.

Example 3: Consider the schema $\mathcal{R} = \{r_1^{io}(A, B), r_2^{io}(B, C), r_3^{io}(C, A)\}$ and the CQ over \mathcal{R} $q(C) \leftarrow r_1(a, B), r_2(B, C)$. We observe that r_3 is not useful to answer the query, i.e., it is irrelevant; let us see why. Using the values obtained from r_2 to access r_3 in order to obtain new values of domain A with which to access r_1 again is pointless. Indeed, the selection on relation r_1 in the query q ensures that the only tuples extracted from r_1 that are those obtained by binding the first argument of r_1 with the value a . ■

Once irrelevant relations are detected, these can be excluded from the access plan, since they will never provide useful bindings for the other relations in order to obtain answers to the query. Since this holds independently of the database instance, cutting on irrelevant relations reduces the number of accesses that need to be made to answer a query.

We now introduce the *dependency graph* (or *d-graph*), a formalism used to characterize the dependencies between input and output arguments in the relations of a schema. This notion will be used to determine relevance of relations.

In order to construct the d-graph, we start from constant-free queries. The elimination of constants from a query requires a preprocessing step: every constant a in the query acts as an *artificial relation* ℓ_a , with a single attribute that is an output attribute, whose content is exactly the tuple $\langle a \rangle$. A constant-free query equivalent to the original one is easily obtained: for example, the query $q(Y) \leftarrow r(a, Y)$ can be replaced by $q(Y) \leftarrow r(X, Y), \ell_a(X)$.

The nodes of a d-graph $G_q^{\mathcal{R}}$ for a query q over a schema \mathcal{R} is determined as follows. Each atom in q corresponds

to a set of nodes (called a *source*) in $G_q^{\mathcal{R}}$, one for each argument of the corresponding relation; such nodes are the *black* nodes. Moreover, for each relation in \mathcal{R} not appearing in q , there is a set of nodes (also called a *source*), one for each argument of the relation; these are the *white* nodes. Each node has two labels: the access mode (“i” or “o”) of the corresponding argument in the relation, and its abstract domain. 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: (i) u and v have the same abstract domain; (ii) u is an output node; (iii) v is an input node. Intuitively, the arcs denote dependencies between relation arguments, indicating that a relation with limited capabilities needs values which can be retrieved from other relations. A source is *free* if none of its nodes has input access mode. Clearly, free sources can be accessed with no restriction.

We indicate with $\text{outArcs}(u, G_q^{\mathcal{R}})$ the set of outgoing arcs from any 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.

With the notion of d-graph at hand, we can define a notion of path for d-graphs, that we denote by *d-path*, which indicates what chains of accesses, starting from free sources, need to be made to access sources that are not free. As shown in the following example, a d-path traverses sources (that are sets of nodes in a d-graph), following arcs that are incoming in a bound node of a source, and arcs that are outgoing from a free node of the same source.

Example 4: Let us consider the schema and query of Example 3. 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 $q(C) \leftarrow r_a(A), r_1(A, B), r_2(B, C)$. The d-graph $G_q^{\mathcal{R}}$ for q is shown in Figure 2, where we have named the sources as the corresponding relations, with a superscript indicating the occurrence number of that relation in the query. The d-graph shows that, starting from the free source r_a , values for the input argument of r_1 can be obtained either directly from r_a or from r_3 via a d-path e_1, e_2, e_3, e_4 . ■

The d-graph gives us also an easily understandable representation of the notions of queryability and answerability: a relation is queryable if and only if all its input nodes are reachable through d-paths that originate from sources having only output attributes. For instance, all relations of Example 4 are queryable, whereas relation r_1 is not queryable w.r.t. query q_2 in Example 2.

D-graphs synthetically represent all possible ways in which values can be obtained from free relations (or chains of relations starting from free ones) to access relations that are not free. The steps of the algorithm of Figure 1 are very similar to a naive execution of all possible accesses indicated in the d-graph. We have seen, however, that irrelevant relations may be part of the schema and that, in general, not all the possible accesses need to be executed to obtain all the answers. Based on these considerations, we now want to

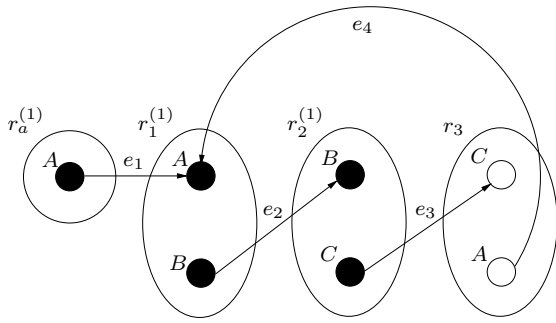


Fig. 2. Dependency graph for Example 4

remove from the d-graph the dependencies that are actually not needed. The overall objective is to generate, from the d-graph after the “pruning”, a query plan that is guaranteed to make only accesses that are necessary for extracting all obtainable answers.

A d-graph may contain arcs not belonging to any d-path that reaches a black node; such arcs may be eliminated. Moreover, we may eliminate some arcs thanks to the presence of joins in the query. We say that an arc $u \rightsquigarrow v$ 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 can 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). A d-graph in which every arc is also labeled as either strong, weak, or deleted is called a *marked d-graph*.

An input node v in a marked d-graph G is inductively defined as *free-reachable* if either (i) there is a weak arc $u \rightsquigarrow v$ in G such that all input nodes in u 's source are free-reachable, or (ii) all strong arcs $u_1 \rightsquigarrow v, \dots, u_n \rightsquigarrow v$ in G are such that all input 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 above-mentioned preprocessing step), a relation maintains its queryability only if all of its input nodes are free-reachable.

Ideally, for a given d-graph, we aim to determine two maximal sets of deleted arcs and strong arcs. 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 $\text{arcs}(G_q^{\mathcal{R}})$ the set of all arcs in d-graph $G_q^{\mathcal{R}}$ and with $\text{cand}(G_q^{\mathcal{R}})$ the set of all candidate strong arcs in $G_q^{\mathcal{R}}$. Clearly, only candidate strong arcs have the potential to become strong arcs. In fact, only those candidate strong arcs that do not destroy free-reachability can actually become strong. In particular, we say that a candidate strong arc γ is *cyclic*, indicated $\text{cycl}(\gamma)$, if it is contained in a cyclic d-path such that all arcs in it are candidate strong; no arc in the set $\text{cycl}(G_q^{\mathcal{R}})$ of cyclic candidate strong arcs of $G_q^{\mathcal{R}}$ can

become strong, since none of its input nodes would be free-reachable. Similarly, no arc in $\text{cycl}(G_q^{\mathcal{R}})$ can be deleted either. Therefore, no candidate strong arc can ever be deleted, since it reaches a black node. We can conclude that the set of strong arcs and the set of deleted arcs must be disjoint.

We call the pair $(\mathcal{S}, \mathcal{D})$ a *solution* for a d-graph $G_q^{\mathcal{R}}$ if \mathcal{S} and \mathcal{D} are respectively sets of strong arcs and deleted arcs (among the arcs of $G_q^{\mathcal{R}}$) that satisfy the above-mentioned conditions; the solution is *maximal* if no other solution $(\mathcal{S}', \mathcal{D}')$ exists such that $\mathcal{S}' \supset \mathcal{S}$ or $\mathcal{D}' \supset \mathcal{D}$. From a solution $(\mathcal{S}, \mathcal{D})$ we indicate with $G_q^{\mathcal{R}(\mathcal{S}, \mathcal{D})}$ the marked d-graph obtained from $G_q^{\mathcal{R}}$ by marking all arcs in \mathcal{D} as “deleted”, all arcs in \mathcal{S} as “strong”, and all remaining arcs as “weak”. Visually, we will remove all arcs labeled as “deleted”, as well as all white 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 any given d-graph. We present in Figure 3 an algorithm that determines such solution by calculating the initial sets of strong arcs (the non-cyclic candidate strong arcs) and deleted arcs (the arcs that are not candidate strong) and then by applying two monotonic fixpoint operators to the sets until a fixpoint is reached. The algorithm has polynomial complexity, as guaranteed by the monotonicity of the fixpoint operators, which are implemented by the functions *unmarkStr* and *unmarkDel* respectively.

The solution calculated by the algorithm indicates the construction of a special marked d-graph, called *optimized d-graph*. Such a graph is important because it straightforwardly provides us with a procedure to determine which sources are relevant: a relation r in a schema \mathcal{R} is relevant for a CQ q over \mathcal{R} iff (i) r is nullary and occurs in q , or (ii) r occurs in the optimized d-graph for q .

Example 5: Consider the schema and the query from example 4, whose d-graph is shown in Figure 2. Arcs e_1 and e_2 are the (non-cyclic) candidate strong arcs and thus they constitute the initial set of strong arcs; arcs e_3 and e_4 are therefore in the initial set of deleted arcs. This is already the greatest fixpoint we were looking for. In particular, arc e_4 remains deleted, since it is dominated by e_1 , which is strong, and then e_3 is deleted as well, since, once e_4 has been deleted, no black node is reachable by a d-path starting with e_3 . The intuition is that relation r_2 does not have to provide arbitrary values to r_3 , and the only value with which we need to access r_1 is a ; indeed, due to the join condition in q , accessing r_1 with values provided by r_3 would not provide tuples that could be used to answer the query q . The optimized d-graph, without deleted arcs, and without source r_3 , is shown in Figure 4; the strong arcs e_1 and e_2 are denoted by a double-lined arrow. ■

IV. GENERATING A MINIMAL QUERY PLAN

In this section, we give a notion of minimality for query plans, based on accesses; then, we present a technique to generate minimal plans from an optimized d-graph.

First of all, we indicate with $\text{Acc}(D, \Pi)$ the *set* of accesses executed by a query plan Π , given a database D . Intuitively,

```

GFP( $G$  : d-graph) : arc set  $\times$  arc set
 $S := \text{cand}(G) \setminus \text{cycl}(G)$ 
 $\mathcal{D} := \text{arcs}(G) \setminus \text{cand}(G)$ 
do {
  ( $S', \mathcal{D}'$ ) := ( $S, \mathcal{D}$ )
   $S := \text{unmarkStr}(S', \mathcal{D}', G)$ 
   $\mathcal{D} := \text{unmarkDel}(S', \mathcal{D}', G)$ 
} while ( $S, \mathcal{D}$ )  $\neq$  ( $S', \mathcal{D}'$ )
return ( $S', \mathcal{D}'$ )

```

```

unmarkStr( $S$  : arc set,  $\mathcal{D}$  : arc set,  $G$  : d-graph) : arc set
 $S' := S$ 
for each arc  $u \hat{\sim} v \in S$ 
  for each arc  $\gamma \in \text{outArCs}(v, G)$ 
    if ( $\gamma \notin S \cup \mathcal{D}$ )  $S' := S' \setminus \{u \hat{\sim} v\}$ ; break
return  $S'$ 

```

```

unmarkDel( $S$  : arc set,  $\mathcal{D}$  : arc set,  $G$  : d-graph) : arc set
 $\mathcal{D}' := \mathcal{D}$ 
for each arc  $u \hat{\sim} v \in \mathcal{D}$ 
  if ( $\text{black}(v)$ ) then
    bool strongExists := false
    for each arc  $u' \hat{\sim} v' \in S$ 
      if ( $v = v'$ ) then strongExists := true; break
    if (not strongExists) then  $\mathcal{D}' := \mathcal{D}' \setminus \{u \hat{\sim} v\}$ 
  else ( $v$  is white)
    if ( $\text{outArCs}(v, G) \setminus \mathcal{D} \neq \emptyset$ ) then  $\mathcal{D}' := \mathcal{D}' \setminus \{u \hat{\sim} v\}$ 
return  $\mathcal{D}'$ 

```

Fig. 3. Implementation of function *GFP*

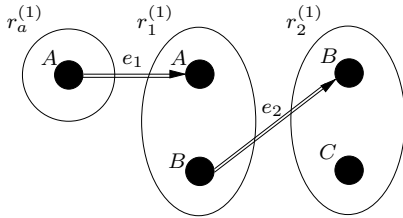


Fig. 4. optimized d-graph for Example 5

our goal would be to find plans that execute the least set of accesses, whatever the database instance; this idea is captured by the following notion of minimality, that we call \forall -minimality. A query plan Π for a CQ q over a schema \mathcal{R} is \forall -minimal iff, for each instance D of \mathcal{R} and for each query plan Π' of q , $\text{Acc}(D, \Pi) \subseteq \text{Acc}(D, \Pi')$.

We are not in luck with this notion, since it is not difficult to prove that, of course depending on the given query, there are cases in which such a minimal plan does not exist.

Example 6: Consider the query $q(X) \leftarrow r_1(X), r_2(Y)$ over the schema $\{r_1^o(A), r_2^o(B)\}$. Any query plan Π must either first access r_1 or r_2 , and, by determinism, always make the same first access, independently of the instance. Suppose Π accesses r_1 first. Consider an instance D in which $r_2 = \emptyset$ and $r_1 = \{a\}$; in D , Π , after accessing r_1 , must access r_2 to determine that $q^D = \emptyset$. But then Π is not \forall -minimal,

since another query plan that accesses r_2 first could realize that $q^D = \emptyset$ without accessing r_1 at all. Symmetrically, no query plan that accesses r_2 first can be \forall -minimal. Therefore, q admits no \forall -minimal query plan. Note that, in our setting, source relations are possibly remote and out of control of the system that queries them; therefore metadata carrying information about the sources (such as the number of tuples they contain) cannot in general be assumed. Note also that extracting the first row from a source does not circumvent this problem, since it corresponds to making an access to that source. ■

We therefore define a less strong notion of minimality; we shall prove that, according to this new notion, there always exists a minimal plan. Let Π and Π' be two query plans for a CQ q over a schema \mathcal{R} . We write $\Pi' \subset \Pi$ whenever, for every instance D , $\text{Acc}(D, \Pi') \subseteq \text{Acc}(D, \Pi)$ and there is an instance D' such that $\text{Acc}(D', \Pi') \subset \text{Acc}(D', \Pi)$. A query plan Π is \subset -minimal iff there is no query plan Π'' for q such that $\Pi'' \subset \Pi$.

Accordingly, a query plan Π is \subset -minimal if there is no other query plan that is strictly better at making accesses than Π . For every query q , there always exists a \subset -minimal query plan, whereas, interestingly, a \forall -minimal query plan for q exists iff the \subset -minimal query plan for q is unique, and in this case they coincide; \subset -minimal query plans represent then the best we can hope for.

Next, we show how to construct a \subset -minimal query plan for a given CQ. We first recall that a CQ q_1 is said to be *minimal* if there is no equivalent CQ q_2 whose body atoms are a subset of q_1 's body atoms. We therefore assume to start from a minimal CQ, knowing that the problem of finding the minimal equivalent of a CQ is NP-complete [5]. From the minimal CQ's optimized d-graph, we are able to derive a \subset -minimal plan, expressed as a Datalog program. The Datalog program is then evaluated according to Datalog's usual least fixpoint semantics, with a few extra expedients that allow stopping the evaluation as soon as possible so as to guarantee access minimality.

If the optimized d-graph refers to more than one source, some relations must necessarily be accessed before others. The order of the accesses to the different relations may either be arbitrary, depending on the database instance, or mandatory, due to the access limitations on the schema. We therefore need to identify which sources must be accessed before other sources. To this end, we determine an ordering between groups of sources in the schema.

Let us denote by $\text{src}(u)$ the source corresponding to a node u in a d-graph. Then, for an optimized d-graph G , we establish an ordering among all the sources such that (i) if there is a weak arc $u \hat{\sim} v$ in G , then $\text{src}(u) \preceq \text{src}(v)$; (ii) if there is a strong arc $u \sim v$ in G , then $\text{src}(u) \prec \text{src}(v)$; (iii) sources traversed by a cyclic d-path have the same order as the other sources in the cycle; all sources outside the cycle have a different order. The same procedure can be applied to the relations corresponding to the sources, with the proviso that there may be no consistent ordering for the relations,

whereas there is always a consistent ordering for the sources in G . The number of possible orderings has an impact on \forall -minimality, since a \forall -minimal query plan exists iff exactly one ordering for the relations is possible. This indicates that, as soon as there are different ways of accessing the relations, \forall -minimality is lost no matter what ordering is chosen, since it is always possible to find a database instance in which failure of the query (i.e., an empty answer) can be determined faster with another ordering. Once an ordering is set and k groups of sources with different orders are determined, we assign a number $pos(s)$ from 1 to k to each source s such that $pos(s_i) < pos(s_j)$ iff $s_i \prec s_j$.

A Datalog program implementing a \subset -minimal query plan for an answerable query can be constructed in the following manner, based on the optimized d-graph as well as on the ordering. First, the original query is rewritten by replacing each atom in the body by an atom with the same arguments but having a new relation name. In particular, for each predicate r in the body of the query, we introduce a new predicate that acts as a sort of *cache* for r in which we store, during the query answering process, all the tuples extracted from r . Note that the name of the cache uniquely identifies the corresponding source, so different occurrences of the same predicate in the query give rise to different cache names; in the following, we choose to add a hat symbol and an occurrence number.

Each cache relation is defined as the corresponding original relation, with additional providers of values for each of its input arguments. A cache relation has the form

$$\hat{r}(I_1, \dots, I_n, O_1, \dots, O_m) \leftarrow r(I_1, \dots, I_n, O_1, \dots, O_m), s^{\hat{r}}_1(I_1), \dots, s^{\hat{r}}_n(I_n)$$

assuming, w.l.o.g., that the first n arguments in r are input arguments and the remaining ones are output arguments. Each $s^{\hat{r}}_i$ is a new predicate name; it names a “domain” relation created to provide values to the corresponding input argument. Such relation takes into account the arcs in the d-graph: if the corresponding incoming arcs are weak, then the relation is defined as a disjunction of the cache relations corresponding to the origin nodes, since any of them can provide input values. Otherwise the relation is defined as a conjunction of those cache relations, since only their join can provide input values.

Finally, the program generated as described above is completed by adding a fact for each artificial 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 7: From the optimized d-graph for q from Example 4, we generate the following code, where the only possible ordering is $\hat{r}_a^{(1)} \prec \hat{r}_1^{(1)} \prec \hat{r}_2^{(1)}$.

$$\begin{aligned} q(X) &\leftarrow \hat{r}_a^{(1)}(A), \hat{r}_1^{(1)}(A, B), \hat{r}_2^{(1)}(B, C) \\ r_a^{(1)}(A) &\leftarrow r_a(A) \\ \hat{r}_1^{(1)}(A, B) &\leftarrow r_1(A, B), s^A(A) \\ \hat{r}_2^{(1)}(B, C) &\leftarrow r_2(B, C), s^B(B) \\ s^A(A) &\leftarrow \hat{r}_a^{(1)}(A) \\ s^B(B) &\leftarrow \hat{r}_1^{(1)}(A, B) \\ r_a(a) &\leftarrow \end{aligned}$$

Here the support relation s^A is defined as $\hat{r}_a^{(1)}$, since there is only one corresponding incoming strong arc; similarly for s^B . Note that the Datalog program above also reflects the fact that r_3 , which could in principle provide useful values to r_1 , is irrelevant and thus not used in the program. ■

We now specify the execution strategy, called *fast-failing strategy*, for such a Datalog program that avoids redundant accesses. For convenience, let us call i -caches the caches corresponding to sources of position i . For each position i from 1 to k in the ordering, we fully populate the i -caches, as described below; finally we evaluate the query over the caches. In order to populate the i -caches, we check whether the subquery including only the j -caches, with $j < i$, is satisfiable in the database. This is done as soon as all j -caches are fully populated: an early non-emptiness test can then be performed on the CQ by checking its conditions only on such caches. If this satisfiability check for the i -caches fails, we exit and report the empty answer, since some of the caches in the query are definitely empty or some of the joins in the query are already known to fail. Else, all their rules are evaluated (they may only refer to j -caches, with $j \leq i$) until a fixpoint is reached, i.e., until no new tuple is extracted for any of the i -caches; in each such rule, the relation is accessed only if all the other conditions succeed. Besides, since there may be several sources for the same relation, we have to make sure to not repeat any access to a relation. For this purpose, we keep track of all access tuples used to access relations; in order to do that, Toorjah uses, for each relation, a sort of “meta-cache” that is defined as the union of all the caches on that relation. Then, before accessing a relation for the evaluation of a cache rule, we check whether the access was already made by consulting its meta-cache. If so, we read the extraction from the corresponding cache; else we make the access proper. Note that accessing a cache has no cost wrt. the notions of minimality introduced in this section.

The previous strategy also indicates that, in case several orderings are possible, a good heuristics may be to choose those orderings that place sources that are involved in more joins first, since they are more likely to lead to failure; for the same reason, if statistical information on the tables is available, it may be advisable, compatibly with the ordering, to place small tables first.

The fast-failing strategy is guaranteed to always calculate the same answer as the fixpoint semantics for the Datalog program, but the former will never repeat an access to a relation and will stop as soon as the answer is known to be

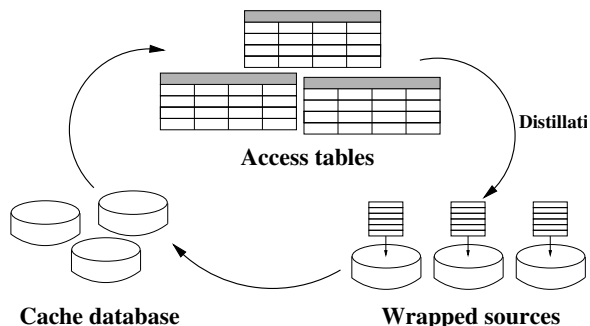


Fig. 5. Data extraction in Toorjah

empty, thus possibly avoiding further accesses that could be made by the latter. This gives us a \subset -minimal query plan.

V. EXPERIMENTAL EVALUATION

We now present the results of the experiments that we carried out with our prototype system Toorjah, that is able to evaluate conjunctive queries over sources with access limitations, using our access minimization strategy. While the query answering algorithm is known, in order to make it work in practice, our implementation requires taking into account several other aspects. In particular, the extraction process of tuples from sources to the caches is realized in Toorjah as depicted in Figure 5, where the following elements can be identified:

- The *cache database* (CDB) stores tuples retrieved during data extraction for a certain query. It consists of physical tables, one for each source table defined in the schema.
- *Access tables* are used to store tuples of values, called *access tuples*, with which to access relations; such tuples are constructed according to the minimal query plan as described in Section IV. There is one access table for each source with limitations.
- *Wrappers* wrap data sources (relational and non-relational), accomplishing the task of querying them in a suitable way. Every wrapper has a queue, where access tuples delivered to it wait to be sent to the corresponding data source.

In Toorjah, we adopt the following strategy to improve query answering. Toorjah tries to access in parallel as many sources as possible. To do so, as soon as an access tuple can be generated from the CDB and sent to a wrapper associated with a subsequent source in the ordering and whose queue is not full, this is done so as to retrieve answers as early as possible. This finely-controlled flow of tuples is labeled with “distillation” in the figure. As mentioned, Toorjah takes also care that every access tuple is never sent twice to the same wrapper.

While query answering in this context is inherently costly, due to the usually high number of accesses to sources, in our experiments we have noticed that the system retrieves tuples (and values) that are significant for the answer in a time that is usually very short, compared to the total execution time. For

this reason, Toorjah presents the result tuples incrementally, as soon as they are generated; this is particularly suitable when the results are paginated. Therefore, the user can interactively stop the lengthy answering process, once (s)he is satisfied with the answers.

We now show experimental results to validate the optimization technique of Toorjah. We have built a prototype system with a query plan optimizer; the relational sources are local, and we measure the number of accesses. Accesses are straightforwardly translated into SQL queries.

For our first series of tests we adopted a fixed schema, reported below: pub_1 and pub_2 store published papers and their authors; $conf$ stores information about papers published in conferences, with the year of publication; rev stores data about reviewers of conferences in certain years; sub stores information about submitted papers and their authors; rev_icde stores information about reviewers of ICDE papers, with the associated evaluation.

$$\begin{aligned}
 &pub_1^{io}(Paper, Person) \\
 &pub_2^{oo}(Paper, Person) \\
 &conf^{ooo}(Paper, ConfName, Year) \\
 &rev^{ooi}(Person, ConfName, Year) \\
 &sub^{oi}(Paper, Person) \\
 &rev_icde^{iio}(Person, Paper, Eval)
 \end{aligned}$$

We populated the above schema with synthetic data, automatically generated. We utilized constants from abstract domains having between 100 and 1000 values. Finally, we randomly populated every source with approximately 1000 tuples. Initially, we considered the following queries:

- 1) $q_1(R) \leftarrow pub_1(P, R), conf(P, C, Y), rev(R, C, Y)$, asking for authors of publications in conferences where they were also reviewers.
- 2) $q_2(R) \leftarrow rev_icde(R, P, rej), conf(P, C, Y), rev(R, C, Y)$, asking for papers rejected at ICDE by a reviewer and then accepted in a conference listing the same reviewer.
- 3) $q_3(R) \leftarrow rev_icde(R, S, acc), sub(S, A), pub_1(P, R), pub_1(P, A), rev(R, icde, 2008), conf(P, icde, Y)$, asking for reviewers of ICDE 2008 who have accepted at ICDE a submission authored by an ICDE coauthor.

In Figure 6 we report, for every query and for every relation, the number of accesses and the number of extracted tuples, first for the naive query plan, and then for the optimized one; when a relation is not included in the plan because it is not relevant, we leave the corresponding cells blank in the table. The improvement in the efficiency due to our optimization has proved to be significant – in the optimized plans many relations are not accessed at all. Notice that it is not surprising that sometimes we get the same values for the same relation for different queries, since such values heavily depend on the constants that the relations have in common, that do not depend on the query. Needless to say, the optimized plans return the same answers as the non-optimized ones. The optimization led to significant pruning of the queries’ d-graphs, that we show in Figure 7, 8, and 9 for q_1 , q_2 , and, resp., q_3 , before and after the pruning. Attribute nodes in the same

relation	q_1				q_2				q_3			
	accesses		returned rows		accesses		returned rows		accesses		returned rows	
	naive	opt.	naive	opt.	naive	opt.	naive	opt.	naive	opt.	naive	opt.
pub_1	4		996		4		996		4		996	
pub_2	399	364	991	884	399		991		399	364	991	884
$conf$	4	1	1000	1000	4	1	1000	1000	4	1	1000	1000
rev	20	20	999	999	20	20	999	999	20	1	999	56
sub	400		996		400		996		400	357	996	893
rev_{icde}	159,600		997		159,600	133,588	997	818	159,600	17,184	997	102

Fig. 6. Experimental results for the test queries

source relations are grouped in an oval; they appear in the same order (top to bottom) as the arguments in the schema (left to right). The relation name is written below the source, with the occurrence number superscripted in brackets. Strong arcs are represented by double-lined arrows, and weak arcs by single-lined arrows.

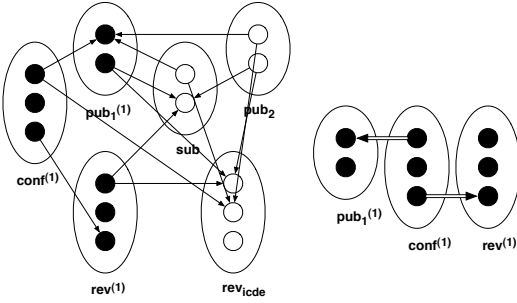


Fig. 7. D-graph and opt. d-graph for q_1

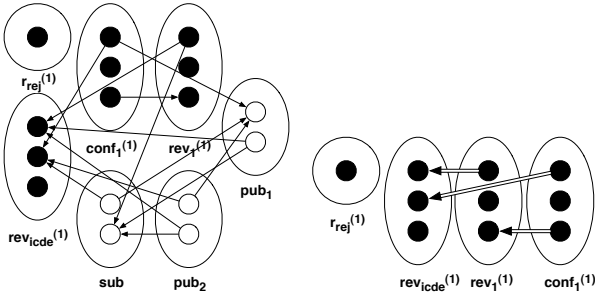


Fig. 8. D-graph and opt. d-graph for q_2

We also tested our approach on randomly generated schemata and queries, with a total of 100 schemata and 100 queries per schema. Each schema comprises 5 to 10 relations; each relation has between 1 and 5 attributes (some of which may have input mode); each of the 10,000 queries has between 2 to 6 atoms and contains at least one join. We considered 100 different database instances in which each relation has between 10 and 10,000 tuples. For fairness, we have excluded two extreme cases: non-answerable queries (for which our algorithm would immediately return the empty answer) and queries on free relations only (for which our algorithm would delete all arcs in the graph, while the naive approach would do a lot of useless work). Figure 10 reports some aggregate data on these experiments, which look very promising. In

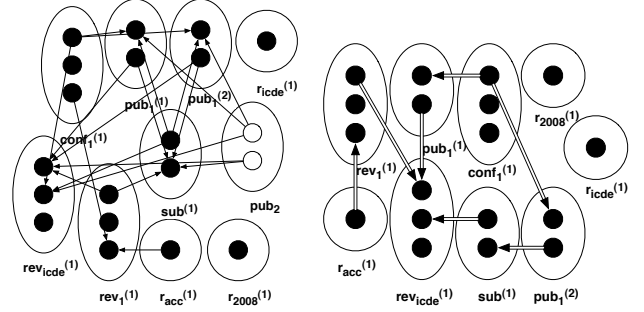


Fig. 9. D-graph and opt. d-graph for q_3

	arcs	deleted arcs	strong arcs	saved accesses
min	10	4	0	9.10%
max	66	65	7	99.99%
avg	20.54	16.23	1.89	81.02%

Fig. 10. Experiments on synthetic queries

particular, we show the minimum, maximum, and average number of: arcs in the d-graph, deleted arcs, and strong arcs in the optimized d-graph. We also show the percentage of accesses avoided thanks to our optimization, which is 81.02% in average; this confirms that the improvement is indeed significant.

Figure 11 reports the average execution times, measured under PostgreSQL on a 3.0GHz quad-core Intel-based machine with 4GB of RAM, for the 10,000 synthetic queries, both with the naive and with the optimized approach, and aggregates them by number of atoms in the query. This confirms, as expected, that the number of accesses heavily weighs upon the execution time, and, therefore, that our optimization is practically relevant. Moreover, one should note that the time needed to retrieve the first answers, promptly displayed by Toorjah thanks to its distillation-based parallelization strategy, is only a small fraction of the total query execution time, and therefore makes the system particularly suited for contexts in which the results can be presented in a paginated manner.

In our tests, if several ordering of the sources were possible, we chose one arbitrarily. Although one could envisage further optimizations depending on the choice of a particular ordering, based, e.g., on statistical information about the sources, such as sampled response time, expected table sizes, etc., these optimizations are, by their nature, highly depending on the instance. Their impact, according to the results shown in this section, can only be relatively small, especially in a context

atoms	naive	opt.
2	9,310 ms	684 ms
3	12,161 ms	1,732 ms
4	10,198 ms	959 ms
5	14,879 ms	1,134 ms
6	15,474 ms	1,247 ms

Fig. 11. Average query execution times

where the answer is paginated.

VI. RELATED WORK

The issue of processing queries under access limitations has been widely investigated in the literature [2], [3], [6], [7]; in particular, [6] considers the optimization of non-recursive plans, [7] addresses the problem of query answering using views, and [2] presents a polynomial-time algorithm to decide whether a conjunctive query can be answered in the presence of access limitations. Recursive query plans were introduced in [8], [3]; in particular, [8] addresses the problem of query containment under access limitations.

Optimizations that can be made during query plan generation, under access limitations, are discussed in [3], [9], [4]. However, all of these works consider a narrow class of queries, named *connection queries*, that is a proper subset of the class of UCQs, that does not include CQs, and that is unfit to express many real-world queries. In a connection query, the attributes with the same abstract domain must be all in join, and they must also be either all selected (with a constant) or all non-selected. Connection queries are evidently inexpressive: take a binary relation *parent* with attributes over the abstract domain *Person*. Unlike CQs, with a connection query one can *only* ask for those who are parents of themselves (or, with a ground query, whether a given person is parent of him/herself). In our experimentation, approximately 70% of our 10,000 synthetically generated queries are not connection queries (and, for instance, also the non-synthetic query q_3 is not a connection query). In the case of connection queries, we are able to determine the same relevant sources as the technique in [4] would, and in addition we further optimize w.r.t. the number of accesses. The technique of [4] does not extend straightforwardly to CQs, since the structure of the joins in a connection query is much simpler than that of a general CQ.

In [10], 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. [11] addresses the problem of ordering subgoals for non-recursive Datalog queries in order to make the query executable from left to right complying with the access limitations. In [12], a run-time optimization technique, that exploits the information about database dependencies that hold on the sources, is presented. [13] solves the (quite general) problem of query answering using views [14] under integrity constraints and under access limitations by reducing it to the same problem under integrity constraints only; various extensions to the query languages are

provided. However, the optimization problem (and, *a fortiori*, the access minimization problem) is not addressed. In [15], the authors analyze the complexity of determining the *feasibility* of a query, i.e., determining whether there exists an equivalent query that is executable as is, while respecting the access limitations. In addition, [15] also discusses the notion of orderability, which is aimed at practical approximations of feasibility for query planning. [16] studies the complexity of the feasibility problem for various query classes.

Some of the results described in this paper have been presented in the informal publication [17].

VII. CONCLUSION

In this paper, we have presented Toorjah, a system that is able to find all obtainable tuples in the answer to a conjunctive query under access limitations. Toorjah evaluates such queries according to a strategy that ensures minimality of the number of accesses to sources, and parallelizes them as much as possible. Since query execution time may be high, the system attempts to obtain answers as early as possible, thus presenting them to the user as they arrive. While further optimizations may be adopted, based, e.g., on estimates of the different response times of the sources or on their expected sizes, our minimization of accesses remains major as well as a necessary step towards efficiency, since all accesses that are unnecessary for all databases are to be excluded anyway. Finally, we have given experimental evidence of the effectiveness of the system by thoroughly testing it against thousands of queries.

Our technique has been extended and proved to be also applicable to more expressive query classes including UCQs with safe negation [18].

Future investigations that may have an immediate impact on query optimization in this context include algorithms for checking query containment under access limitations.

ACKNOWLEDGMENTS

Andrea Cali was partially supported by the EPSRC project “Schema Mappings and Automated Services for Data Integration and Exchange” (EP/E010865/1). Davide Martinenghi acknowledges support from Italian PRIN project “New technologies and tools for the integration of Web search services”. The authors wish to thank Domenico Carbotta for his valuable contribution to the experimental evaluation of the system, and Diego Calvanese for his precious comments during the initial development of this work.

REFERENCES

- [1] D. Florescu, A. Levy, and A. Mendelzon, “Database techniques for the World-Wide Web: A survey,” *SIGMOD Record*, vol. 27, no. 3, pp. 59–74, 1998.
- [2] A. Rajaraman, Y. Sagiv, and J. D. Ullman, “Answering queries using templates with binding patterns,” in *Proc. of PODS’95*, 1995.
- [3] C. Li and E. Chang, “Query planning with limited source capabilities,” in *Proc. of ICDE 2000*, 2000, pp. 401–412.
- [4] —, “Answering queries with useful bindings,” *ACM Trans. on Database Systems*, vol. 26, no. 3, pp. 313–343, 2001.
- [5] A. K. Chandra and P. M. Merlin, “Optimal implementation of conjunctive queries in relational data bases,” in *Proc. of STOC’77*, 1977, pp. 77–90.

- [6] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciuc, "Query optimization in the presence of limited access patterns," in *Proc. of ACM SIGMOD*, 1999, pp. 311–322.
- [7] O. M. Duschka and A. Y. Levy, "Recursive plans for information gathering," in *Proc. of IJCAI'97*, 1997, pp. 778–784.
- [8] T. D. Millstein, A. Y. Levy, and M. Friedman, "Query containment for data integration systems," in *Proc. of PODS 2000*, 2000, pp. 67–75.
- [9] C. Li and E. Chang, "On answering queries in the presence of limited access patterns," in *Proc. of ICDT 2001*, 2001, pp. 219–233.
- [10] C. Li, "Computing complete answers to queries in the presence of limited access patterns," *VLDB Journal*, vol. 12, no. 3, pp. 211–227, 2003.
- [11] G. Yang, M. Kifer, and V. K. Chaudhri, "Efficiently ordering subgoals with access constraints," in *Proc. of PODS 2006*, 2006, pp. 22–22.
- [12] A. Cali and D. 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)*. Kluwer Academic Publishers, 2002, pp. 285–301.
- [13] A. Deutsch, B. Ludäscher, and A. Nash, "Rewriting queries using views with access patterns under integrity constraints," in *Proc. of ICDT 2005*, 2005, pp. 352–367.
- [14] A. Y. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.
- [15] B. Ludäscher and A. Nash, "Processing first-order queries under limited access patterns," in *Proc. of PODS 2004*, 2004, pp. 307–318.
- [16] —, "Processing union of conjunctive queries with negation under limited access patterns," in *Proc. of EDBT 2004*, 2004, pp. 422–440.
- [17] A. Cali, D. Calvanese, and D. Martinenghi, "Optimization of query plans in the presence of access limitations," in *EROW 2007 (ICDT workshop)*, M. Arenas and J. Hidders, Eds. Informal proceedings, 2007, pp. 33–47.
- [18] A. Cali and D. Martinenghi, "Determining relevant sources in conjunctive query answering under access limitations." Submitted, 2008.