

Modeling Functional Data Sources as Relations

Simone Santini and Amarnath Gupta*

University of California, San Diego

Abstract. In this paper we present a model of functional access to data that, we argue, is suitable for modeling a class of data repositories characterized by functional access, such as web sites. We discuss the problem of modeling such data sources as a set of relations, of determining whether a given query expressed on these relations can be translated into a combination of functions defined by the data sources, and of finding an optimal plan to do so.

We show that, if the data source is modeled as a single relation, an optimal plan can be found in a time linear in the number of functions in the source but, if the source is modeled as a number of relations that can be joined, finding the optimal plan is NP-hard.

1 Introduction

These days, we see a great diversification in the type, structure, and functionality of the data repositories with which we have to deal, at least when compared with as little as fifteen or twenty years ago. Not too long ago, one could quite safely assume that almost all the data that a program had both the need and the possibility to access were stored in a relational database or, were this not the case, that the amount of data, their stability, and their format made their insertion into a relational database feasible.

As of today, such a statement would be quite undefensible. A large share of the responsibility for this state of affairs must be ascribed, of course, to the rapid diffusion of data communication networks, which created a very large collection of data that a person or a program might want to use. Most of the data available on data communication networks, however, are not in relational form [1] and, due to the volume and the instability of the medium, the idea of storing them all into a stable repository is quite unfeasible.

The most widely known data access environment of today, the *world-wide web*, was created with the idea of displaying reasonably well formatted pages of material to people, and of letting them “jump” from one page to another. It followed, in other words, a rather procedural model, in which elements of the page definition language (*tags*) often stood for actions: a link specified a “jump” from one page to another. While a link establishes a connection between two pages, this connection is not symmetric (a link that carries you from page *A* to page *B* will not carry you from page *B* to page *A*) and therefore is not a *relation* between two pages (in the sense in which the term

* The work presented in this paper was done under the auspices and with the funding of NIH project NCRR RR08 605, *Biomedical Informatics Research Network*, which the authors gratefully acknowledge.

“relation” is used in databases), but rather a *functional connection* that, given page *A*, will produce page *B*.

In addition to this basic mechanism, today many web sites that contain a lot of data allow one to specify search criteria using the so-called *forms*. A form is an input device through which a fixed set of values can be assigned to an equally fixed set of parameters, the values forming a search criterion against which the data in the web site will be matched, returning the data that satisfy the criterion.

Consider the web site of a public library (an example to which we will return in the following). Here one can have a form that, given the name of an author returns a *web page* (or other data structures) containing the titles of the books written by that author. This doesn't imply that a corresponding form will exist that, given the title of a book, will return its author. In other words, the dependence author \longrightarrow book is not necessarily invertible. This limitation tells us that we are not in the presence of a set of relations but, rather, in the presence of a data repository with *functional access*. The diffusion of the internet as a source of data has, of course, generated a great interest in the conceptual modeling of web sites [2–4]. In this paper we present a formalization of the problem of representing a functional data source as a set of relations, and of translating (whenever possible) relational queries into sequences of functions.

2 The Model

For the purpose of this work, a *functional data source* is a set of procedures that, given a number of attributes whose value has been fixed, instructs us on how to obtain a data structure containing further attributes related to the former.

To fix the ideas, consider again the web site of a library. A procedure is defined that, given the name of an author, retrieves a data structure containing the titles of all the books written by that author. The procedure for doing so looks something like this:

Procedure 1: author \rightarrow set(title)

- i) go to the “search by author” page;
- ii) put the desired name into the “author” slot of the form that you find there;
- iii) press the button labeled “go”;
- iv) look at the page that will be displayed next, and retrieve the list of titles.

Getting the publisher and the year of publication of a book, given its author and title is a bit more complicated:

Procedure 2: author, title \rightarrow publisher, year

- i) execute procedure 1 and get a list of titles;
- ii) search the desired title in the list;
- iii) if found then
 - iii.1) access the book page, by clicking on the title;
 - iii.2) search the publisher and year, and return them;
- iv) else fail.

On the other hand, in most library web pages there is no procedure that allows one to obtain a list of all the books published by a given publisher in a given year, and a query asking for such information would be impossible to answer.

We start by giving an auxiliary definition, and then we give the definition of the kind of functional data sources that we will consider in the rest of the paper.

Definition 1. A data sort S is a pair (N, T) , written $S = N : T$, where N is the name of the sort, and T its type. Two data sorts are equal if their names and their types coincide.

A data sort, in the sense in which we use the term here, is not quite a “physical” data type. For instance, *author:string* and *title:string* are both of the same data type (string) but they represent different data sorts¹. The set of *complex sorts* is the transitive closure of the set of data sorts with respect to Cartesian product of sorts and the formation of collection types (sets, bags, and lists).

Definition 2. A functional data source is a pair (S, F) where $S = \{S_1, \dots, S_n\}$ is a set of data sorts, $F = \{f_1, \dots, f_m\}$ is a set of functions $\alpha \xrightarrow{f} \beta$, where both α and β are composite sorts made of sorts in S .

In the library web site, *author:string*, and *year:int* are examples of data sorts. The procedures are instantiations of functions. Procedure 1, for example, instantiates a function $\text{author:string} \xrightarrow{f} \text{title:string}$.

The elements “author:string” and “title:string” are examples of composite sorts. Sometimes, when there is no possibility of confusion, we will omit the type of the sort. Our goal in this paper is to model a functional data source like this one in a way that resembles a set of relations upon which we can express our query conditions. To this end, we give the following definition.

Definition 3. A relational model of a functional data source is a set of relations $R = \{R_1, \dots, R_p\}$ where $R_i \subseteq S_{i_1} \times \dots \times S_{i_q}$ and all the S_i ’s are sorts of the functional data source. The relation R_i is called a relational façade for the underlying data source, and will sometimes be indicated as $R_i(N_{i_1} : T_{i_1}, \dots, N_{i_q} : T_{i_q})$.

The problems we consider in this paper are the following: (1) Given a model R of a functional data source (S, F) and a query on the model, is it possible to answer the query using the procedures f_i defined for the functional data source?, and (2) if the answer to the previous question is “yes,” is it possible to find an optimal sequence of procedures that will answer the query with minimal cost?

It goes without saying that not all the queries that are possible on the model are also possible on the data source. Consider again the library web site; a simple model for this data source is composed of a single relation, that we can call “book,” and defined as:

book(name:string, title:string, publisher:string, year:int).

¹ The entities that we call data sorts are known in other quarters as “semantic data types.” This name, however, entails a considerable epistemological commitment, quite out of place for a concept that, all in all, has nothing semantic about it: an *author:string* is as syntactic an entity as any abstract data type, and does not require extravagant semantic connotations.

A query like

$(N, T) :- \text{book}(N, T, \text{'dover'}, 1997),$

asking for the author and title of all books published by dover in 1997 is quite reasonable in the model, but there are no procedures on the web site to execute it.

We will assume, to begin with, that the model of the web site contains a single relation. In this case we can also assume, without loss of generality, that the relation is defined in the Cartesian product of all the sorts in the functional data source: $R(S_1, \dots, S_n)$. Throughout this paper, we will only consider non-recursive queries. It should be clear in the following that recursive queries require a certain extension of our method, but not a complete overhaul of it. Also, we will consider conjunctive queries², whose general form can be written as:

$$(S_{k_1}, \dots, S_{k_p}) : -R(S_1, \dots, S_n), S_{j_1} = c_1, \dots, S_{j_q} = c_q, \phi_1(S_{11}, S_{12}), \dots, \phi_1(S_{u1}, S_{u2}) \quad (1)$$

where c_1, \dots, c_q are constants, all the S 's come from the sorts of the relation R , and the ϕ_i 's are comparison operators drawn from a suitable set, say $\phi_i \in \{<, >, =, \neq, \leq, \geq\}$.

We will for the moment assume that the functional data source provides no mechanism for verifying conditions of the type $\phi_1(S_{11}, S_{12})$. The only operations allowed are retrieving data by entering values (constants) in a suitable field of a form or traversing a link in a web site with a constant as a label (such as the title of a book in the library example). Given the query (1) in a data source like this, we would execute it by first determining whether the function $f : S_{j_1} \times \dots \times S_{j_n} \rightarrow \{S_{k_1} \times \dots \times S_{k_p} \times S_{11} \times \dots \times S_{u2}\}$ can be computed. If it can, we compute $f(c_1, \dots, c_q)$ and, for each result returned, check whether the conditions $\phi_i(S_{i1}, S_{i2})$ are verified.

The complicated part of this query schema is the first step: the determination of the function f that, given the constants in the query, allows us to obtain the query outputs $\{S_{k_1}, \dots, S_{k_p}\}$, augmented with all the quantities needed for the comparisons.

3 Query Translation

Informally, the problem that we consider in this section is the following. We have a collection of data sorts $S = \{S_1, \dots, S_n\}$. Given two data sorts α, β , defined as Cartesian products of elements of S ($\alpha = S_{\alpha_1} \times \dots \times S_{\alpha_a}$ and $\beta = S_{\beta_1} \times \dots \times S_{\beta_b}$) one can define a formal (and unique) *correspondence function* $f_{\alpha\beta} : \alpha \rightarrow \beta$. This function operates on the model of the data source (this is why we used the adjective “formal” for it: it is not necessarily a function that one can compute) and, given the values $\{S_{\alpha_1}, \dots, S_{\alpha_a}\}$, returns the corresponding values $\{S_{\beta_1}, \dots, S_{\beta_b}\}$. If $\{v_1, \dots, v_a\}$ are the input values, this function computes the relational algebra operation

$$\pi_{N_{\beta_1}, \dots, N_{\beta_b}} \circ \sigma_{S_{\alpha_1}=v_1, \dots, S_{\alpha_a}=v_a} \quad (2)$$

where the N 's are the names of the sorts S , as per definition 1. A correspondence function can be seen, in other words, as the functional counterpart of the query (2)

² Any query can, of course, be translated in a disjunctive normal form, that is, in a disjunction of conjunctive queries. The system in this case will simply pose all the conjunctive queries and then take the union of all the results.

which, on a single table, is completely general. (Remember that we don't yet consider conditions other than the equality with a constant.)

The set $F = \{f_{\alpha\beta}\}$ of all correspondence functions contains the grounding of all queries that we might ask on the model. The functional data source, on the other hand, has procedures P_i , each one of which implements a specific function $f_{\alpha\beta}$, a situation that we will indicate with $P_i \rightsquigarrow f_{\alpha\beta}$. The set of all implemented correspondence functions is $F|_{\rightsquigarrow} = \{f | \exists P : P \rightsquigarrow f\}$. Our query implementation problem is then, given a query q , with the relative correspondence function f , to find a suitable combination of functions in $F|_{\rightsquigarrow}$ that is equal to f . In order to make this statement more precise, we need to clarify what do we mean by "suitable combination of functions" that is, we need to specify a function algebra. We will limit our algebra to three simple operations that create sequences of functions, as shown in Table 1. (We assume, pragmatically, that more complex manipulations are done by the procedures P_i .)

Table 1. Operators of the function algebra.

Operation	Definition	Description	Typing
$f \circ g$	$(f \circ g)(x) = f(g(x))$	function composition	$\frac{f:\alpha \rightarrow \beta \quad g:\gamma \rightarrow \alpha}{f \circ g:\gamma \rightarrow \beta}$
$\langle f, g \rangle$	$\langle f, g \rangle(x) = (f(x), g(x))$	cartesian composition	$\frac{f:\alpha \rightarrow \beta \quad g:\alpha \rightarrow \gamma}{\langle f, g \rangle:\alpha \rightarrow \beta \times \gamma}$
$f \times g$	$(f \times g)(x, y) = (f(x), g(y))$	cartesian product	$\frac{f:\alpha \rightarrow \beta \quad g:\delta \rightarrow \gamma}{f \times g:\alpha \times \delta \rightarrow \beta \times \gamma}$

A function $f \in F|_{\rightsquigarrow}$ for which a procedure is defined, and that transforms a data sort S into a data sort P can be represented as a diagram

$$S \xrightarrow{f} P. \quad (3)$$

The operators of the function algebra generate diagrams like those in the first and third column of Table 2. In order to obtain the individual data types, we introduce the formal operator of projection. The projection is "formal" in that it exists only in the diagrams: in practice, when we have the data type $P \times Q$ we simply select the portion of it that we need. The projections don't correspond to any procedure and their cost is zero. The dual of the projection operator is the Cartesian product which, given two data of type A and B produces from them a datum of type $A \times B$. This is also a formal operator with zero cost. where the dotted line with the \times symbols is there to remind us that we are using a Cartesian product operator, and the arrow goes from the type that will appear first in the product to the type that will appear second (we will omit the arrow when this indication is superfluous).

The Cartesian product of the functions $S \xrightarrow{f} P$ and $Q \xrightarrow{g} R$ is represented as

$$S \times Q \xrightarrow{f \times g} P \times R \quad (4)$$

With these operations, and the corresponding diagrams, in place, we can arrange the correspondence functions $f \in F|_{\rightsquigarrow}$ in a diagram, which we call the *computation diagram* of a data source.

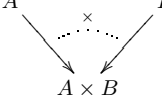
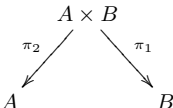
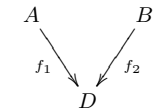
Definition 4. The computation diagram of a functional data source is a graph $G = (N, E)$ with nodes labeled by a labeling function $\lambda_n : N \rightarrow S$, S being the set of composite data sorts of the source, and edges labeled by the labeling function $\lambda_e : E \rightarrow F_{|\rightsquigarrow}$ such that each edge is one of the following:

1. A function edge, such that if the edge is (n_1, n_2) , then $\lambda_E((n_1, n_2)) : \lambda_n(n_1) \rightarrow \lambda_n(n_2)$, and represented as in (3);
2. projection edges,
3. cartesian product edges

Let us go back now to our original problem. We have a query and a correspondence function $S_1 \times \dots \times S_n \xrightarrow{f} P_1 \times \dots \times P_m$ that we need to compute, where S_1, \dots, S_n are the data sorts for which we give values, and P_1, \dots, P_m are the results that we desire. In order to see whether the computation is possible, we adopt the following strategy: first, build the computation diagram of the data source, then we add a node called s to the graph, and connect it to S_1, \dots, S_n , as well as a node d , with edges coming from P_1, \dots, P_m ; finally, we check whether a path exists from s to d .

If we are to find an optimal solution to the grounding of a correspondence function f , we need to assign a cost to each node of the graph and, in order to do this, we need to determine the cost of traversing an edge. The cost functions of the various combinations that appear in a computation graph are defined in table 2.

Table 2. Cost of the functional operations in terms of graph path.

Operation	Cost	Operation	Cost
$A \xrightarrow{f} B$	$C(B) = C(A) + C(f)$		$C(A \times B) = C(A) + C(B)$
	$C(A) = C(B) = C(A \times B)$		$C(D) = \min(C(A) + c(f_1), C(B) + c(f_2))$

The problem of finding the optimal functional expression for a given query can therefore be reduced to that of finding the shortest path in a suitable function graph, a problem that we will now briefly elucidate. Let G be a function graph, $G.V$ the set of its vertices, and $G.E$ the set of its edges.

For every node $u \in G.V$, let $u.\kappa$ be the distance between u and the source of the path, $u.\pi$ the predecessor(s) of u in the minimal path, and $u.\nu$ the set of nodes adjacent to u (accounting for the edge directions)

In addition, a cost function $c : \text{vertex} \times \text{vertex} \rightarrow \text{real}$ is defined such that $c(u, v)$ is the cost of the edge (u, v) . If $(u, v) \notin G.E$, then $c(u, v) = \infty$.

The algorithm in table 3 uses the Dijkstra's shortest path algorithm to build a function graph that produces a given set of output from a given set of input, if such a graph

exists: the function $\text{dijkstra}(G, c, s)$ returns the set of nodes in G where, for each node n , $n.\kappa$ is set to the cost of the path from s to n according to the cost function c . Dijkstra's algorithm is a standard one and is not reported here.

Table 3. Algorithm for the creation of function graphs.

```

make_graph( $I : \{\text{vertex}\}, O : \{\text{vertex}\}, G : \text{graph}, c : \text{vertex} \times \text{vertex} \rightarrow \text{real}$ ) : graph
   $s, d : \text{vertex};$ 
   $G.V := G.V \cup \{s, d\};$ 
  forall  $u$  in  $I$  do  $G.E := G.E \cup \{(s, u)\};$  od
  forall  $u$  in  $O$  do  $G.E := G.E \cup \{(u, d)\};$  od;
   $S := \text{dijkstra}(G, c, s);$ 
   $Q : \text{graph};$ 
   $T, P : \{\text{vertex}\};$ 
   $S := S - \{s, d\}; Q.V := \emptyset; T := O; P := O;$ 
  while  $T \neq \emptyset$  do
     $u := \text{element}(T);$ 
    if  $u.\nu \neq s \wedge u.\nu \neq \emptyset$  do
      forall  $v$  in  $I$  do
         $Q.V := Q.V \cup \{v\};$ 
        if  $v \notin P$  do  $T := T \cup \{v\}$  fi
         $P := P \cup \{v\}; Q.E := Q.E \cup \{(v, u)\};$ 
      od;
     $T := T - \{u\};$ 
  fi;
od;
return  $Q;$ 

```

4 Relaxing Some Assumptions

The model presented so far is a way of solving a well known problem: given a set of functions, determine what other functions can be computed using their combination; our model is somewhat more satisfying from a modeling point of view because of the explicit inclusion of the cartesian product of data sorts and the function algebra operators necessary to take them into account but, from an algorithmic point of view, what we are doing is still finding the transitive closure of a set of functional dependencies. We will now try to ease some of the restrictions on the data source. These extensions, in particular the inclusion of joins, can't be reduced to the transitive closure of a set of functional dependencies, and therein lies, from our point of view, the advantage of the particular form of our model.

Comparisons. The first limitation that we want to relax is the assumption that the data source doesn't have the possibility of expressing any of the predicates $\phi_i(S_{i1}, S_{i2})$ in the query (1). There are cases in which some limited capability in this sense is available.

We will assume that the following limitations are in place: firstly, the data sources provides a finite number of predicate possibilities; secondly each predicate is of the form $\phi(S, R) = S \text{ op } R$, where S and R are fixed data sorts, and “op” is an operator that can be chosen amongst a finite number of alternative. The general idea here comes, of course, from an attempt to model web sites in which conditions can be expressed as part of “forms.”

In order to incorporate these conditions into our method, one can consider them as data sorts: each condition $\phi(S, R)$ is a data sort that takes values in the set of triples (s, r, op) , with s of sort S and r of sort R . In other words, indicating a sort as a pair $N : T$, where N is the name and T the data type of the sort, a comparison data sort $\mathfrak{C}(S_1, S_2)$ is isomorphic to $N_1 : T_1 \times N_2 : T_2 \times (T_1 \times T_2 \rightarrow \mathbf{2})$ where $\mathbf{2}$ is the data type of the booleans. A procedure that accepts in input a value of a data sort S_1 , and a condition on the data sorts S_2, S_3 , would be represented as

$$\begin{array}{ccc}
 S_1 & & \mathfrak{C}(S_2, S_3) \\
 \swarrow & \xrightarrow{\quad \times \quad} & \searrow \\
 & S_1 \times \mathfrak{C}(S_2, S_3) & \\
 & \downarrow f & \\
 & \alpha &
 \end{array} \tag{5}$$

The only difference between condition data sorts and regular data sorts is that conditions can’t be obtained as the result of a procedure, so that in a computation graph a condition should not have any incoming edge.

Joins. Let us consider now the case in which the model of the functional data source consists of a number of relations. We can assume, for the sake of clarity, that there are only two relations in the model:

$$\begin{array}{l}
 R_1(N_1 : T_1, \dots, N_p : T_p) \\
 R_2(M_1 : Q_1, \dots, M_v : Q_v).
 \end{array} \tag{6}$$

Each of these relations supports intra-relational queries that can be translated into functions and executed using the computation graph of that part of the functional source that deals with the data sorts in the relation. In addition, however, we have now queries that need to *join* data between the two relations. Consider the relations: $R_1(X_1, X_2, X_3)$, $R_2(Y_1, Y_2, Y_3)$ and the following query:

$$(A, B) : -R_1(X, 'x', A), R_2(Y, 'y', B), A = B. \tag{7}$$

We can compute this query in two ways. The first makes use of the following two correspondence functions:

$$\begin{array}{l}
 X_2 \xrightarrow{f_1} X_1 \times X_3 \\
 Y_2 \times Y_3 \xrightarrow{f_2} Y_1.
 \end{array} \tag{8}$$

To implement this query, we adopt the following procedure:

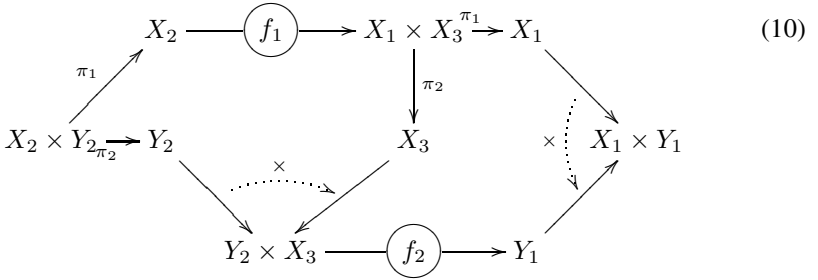
Procedure 3:

- i) use the computation graph of R_1 to compute $f_1('x')$, returning a set of pairs $(a : X_1, b : X_2)$;
- ii) for each pair (a, b) returned:
 - ii.1) compute $f_2('y', b)$ using the graph of R_2 , obtaining a set of results $(c : Y_1)$;
 - ii.2) for each c , form the pair (a, c) , and add it to the output.

The procedure can be represented using a computation graph in which the graphs that compute f_1 and f_2 are used as components. Let us indicate the graph that computes the function f_{as} :

$$\alpha \longrightarrow \textcircled{f} \longrightarrow \beta \quad (9)$$

Then a join like that in the example is computed by the following diagram:



The second possibility to compute the join is symmetric. While in this case we used the relation R_1 to produce the variable on which we want to join and the relation R_2 to impose the join condition, we will now do the reverse. We will use the functions

$$\begin{aligned} Y_2 &\xrightarrow{f_3} Y_1 \times Y_3 \\ X_2 \times X_3 &\xrightarrow{f_4} X_1. \end{aligned} \quad (11)$$

and a computation diagram similar to the previous one. Checking whether the source can process the join, therefore, requires checking if either the pairs of functions (f_1, f_2) or (f_3, f_4) can be computed. The concept can be easily extended to a source with many relations and a query with many joins as follows.

Take a conjunctive query, and let $J = \{J_1, \dots, J_n\}$ the set of its joins, with $J_i : (X_i = Y_i)$. We can always rewrite a query so that each variable X will appear in only one relation, possibly adding some join conditions. Consider, for example, the fragment $R(A, X), P(B, X), Q(C, X)$, which can be rewritten as

$$R(A, X_1), P(B, X_2), Q(C, X_3), X_1 = X_2, X_2 = X_3. \quad (12)$$

We will assume that all queries are normalized in this way. Given a variable X , let $s(X)$ be the relation in which X appear. Also, given a relation R in the query, let $i(R)$ the Cartesian product of its input sorts, and $o(R)$ the Cartesian product of its output sorts.

Table 4. Algorithm for the verification of the join conditions.

```

check( $I : \{\text{vertex}\}, O : \{\text{vertex}\}, G : \text{graph}, c : \text{vertex} \times \text{vertex} \rightarrow \text{real}$ ) : real
   $s, d : \text{vertex};$ 
   $G.V := G.V \cup \{s, d\};$ 
  forall  $u$  in  $I$  do  $G.E := G.E \cup \{(s, u)\}$  od;
  forall  $u$  in  $O$  do  $G.E := G.E \cup \{(u, d)\}$  od;
   $S := \text{dijkstra}(G, c, s);$ 
  return  $(d.\kappa < \infty);$ 

```

The algorithm for query rewriting is composed of two parts. The first is a function that determines whether a function from a given set of input to a given set of outputs can be implemented, and represented in Table 4. The second finds a join combination that satisfies the query. It is assumed that a set of the join conditions that appear in the query $J = \{(X_1, Y_1), \dots, (X_n, Y_n)\}$ is given. The algorithm, reported in table 5 returns a computation graph that computes the query with all the required joins.

Table 5. Join determination algorithm.

```

joins( $J : \{\text{vertex} \times \text{vertex}\}, G : \text{graph}, c : \text{vertex} \times \text{vertex} \rightarrow \text{real}$ ) : graph
1.  $I := \emptyset$ 
2. forall  $(X, Y)$  in  $J$  do
    $R := s(X); Q := s(Y);$ 
   if  $\text{check}(i(R), o(R) \cup \{X\}, G, c) \wedge \text{check}(i(Q) \cup \{Y\}, o(Q), G, c)$  do
      $I := I \cup \{(X, Y)\}$ 
   elseif  $\text{check}(i(Q), o(Q) \cup \{Y\}, G, c) \wedge \text{check}(i(R) \cup \{X\}, o(R), G, c)$  do
      $I := I \cup \{(Y, X)\}$ 
   fi od;
3.  $Q := \text{make\_graph}(\bigcup_i i(R_i) \cup \{X | (X, Y) \in I\}, \bigcup_i o(R_i) \cup \{Y | (X, Y) \in I\}, G, c);$ 
4. forall  $u$  in  $O$  do  $Q.E := Q.E \cup \{(X, Y)\}$  od;
5. if  $\text{cycle}(Q)$  do error fi;
6. return  $Q;$ 

```

The correctness of the algorithm is proven in the following proposition:

Proposition 1. *Algorithm 1 succeeds if and only if the query with the required joins can be executed.*

The proof can be found in [5].

While the algorithm “joins” is an efficient (linear in the number of joins) way of finding a plan whose correctness is guaranteed, finding an *optimal* plan is inherently harder:

Theorem 1. *Finding the minimal set of functions that implements all the joins in the query is NP-hard.*

Proof. We prove the theorem with a reduction from *graph cover*. let $G = (V, E)$ be a graph, with sets of nodes $V = \{v_1, \dots, v_n\}$, edges $E = \{e_1, \dots, e_m\}$, and with

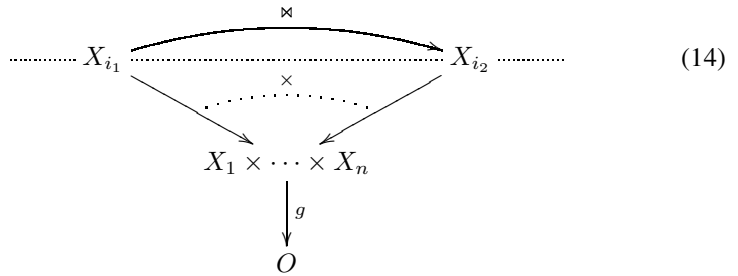
$e_i = (v_{i_1}, v_{i_2})$, $v_{i_1}, v_{i_2} \in V$. Given such a graph, we build a functional source and a query as follows.

For each node v_i define a sort X_i and a function $f : I \rightarrow X_i$. All the sorts are of the same data type. For each edge (v_h, v_k) define a condition $X_h = X_k$. Also, define a function $g : X_1 \times \dots \times X_n \rightarrow Y$. Finally, define the relations $R_1(I, X_1)$, $R_2(X_1, X_2), \dots, R_{n+1}(X_n, Y)$ and the query

$$\text{ans}(Y) : -R_1(I, X_1), R_2(X_1, X_2), \dots, R_{n+1}(X_n, Y), \\ I = 'i', X_{1_1} = X_{1_2}, \dots, X_{m_1} = X_{m_2} \quad (13)$$

where the equality conditions are derived from the edges of the graph. The reduction procedure is clearly polynomial so, in order to prove the theorem we only need to prove that a solution of graph cover for G exists if and only if a cost-bound plan can be found for the query.

1. Suppose that a query plan for the query exists that uses $B + 1$ functions: $P = \{f_1, \dots, f_B, g\}$ (the function g must obviously be part of every plan, since it is the only function that gives us the required output Y). Consider the set $S = \{v_i | f_i \in P\}$, which contains, clearly, B nodes, and the edge (v_{i_1}, v_{i_2}) of the graph. This edge is associated to a condition $X_{i_1} = X_{i_2}$ in the query and, since the query has been successfully planned, either the function f_{i_1} or f_{i_2} are in the plan. Consequently, either v_{i_1} or v_{i_2} are in the set, and the edge (v_{i_1}, v_{i_2}) is covered.
2. let now $S = \{v_1, \dots, v_B\}$ be a covering and consider the plan $P = \{f_i | v_i \in S\} \cup \{g\}$. The output is clearly produced correctly as long as all the join conditions are satisfied. let $X_{i_1} = X_{i_2}$ be a join condition. This corresponds to an edge (v_{i_1}, v_{i_2}) and, since S is a covering, either v_{i_1} or v_{i_2} are in S . Assume that it is v_{i_1} (if it is v_{i_2} we can clearly carry out a similar argument). Then the plan contains the function f_{i_1} , which computes X_{i_1} so that the variables X_{i_1} and X_{i_2} and the join are determined by the following graph fragment

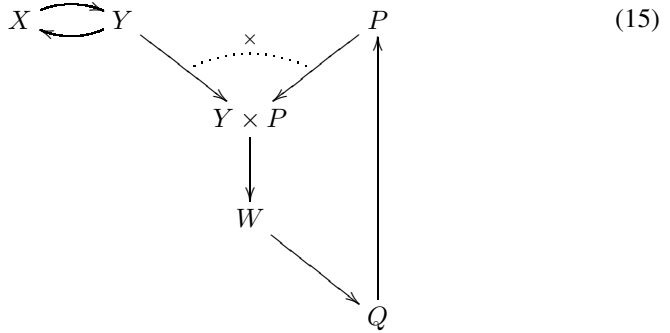


5 Related Work

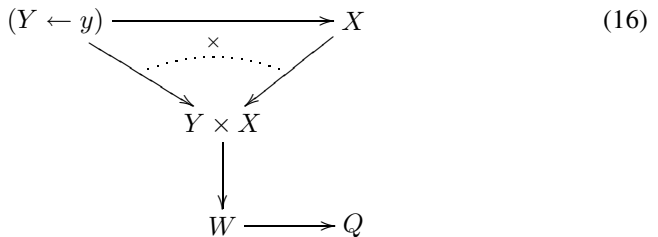
The idea of modeling certain types of functional sources using a relational façade (or some modification thereof) is, of course, not new. The problem of conciliating the broad matching possibilities of a relation with the constraints deriving from the source has been solved in various ways the most common of which, to the best of our knowledge, is by the use of *adornments* [6, 7], which also go under the name of *binding patterns*.

Given a relation $R(X_1, \dots, X_n)$, a binding pattern is a classification of the variables X_1, \dots, X_n into *input* variables (which must be “bound” when the relation is accessed in the query, hence the name of the technique), *output* variables (which must be free when the relation is accessed), and *dyadic* variables, which can be indifferently inputs or outputs. Any query that accesses the relation by assigning values to the input variables and requiring values for some or all the output variables can be executed on that relation façade. A relational façade can, of course, have multiple binding patterns. If the relational façade is used to model an n -ary relation isomorphic to it, for instance, it allows all the 2^n possible bound/free binding patterns on its variables or, equivalently, all its variables are dyadic. In the following, a binding pattern for any n -ary relation will be represented as a string $b \in \{i, o, d\}$ (where i , o , and d stand for *input*, *output*, and *dyadic*, respectively, although dyadic variables will not appear in the examples that follow). Unlike our technique, which determines query feasibility at run time, binding patterns are determined as part of the model. This difference results in a number of limitations of binding patterns, some examples of which are given below.

Multiple relations with hidden sorts. Consider a source with five sorts, X, Y, P, W, Q , and the functional dependencies shown in the following diagram



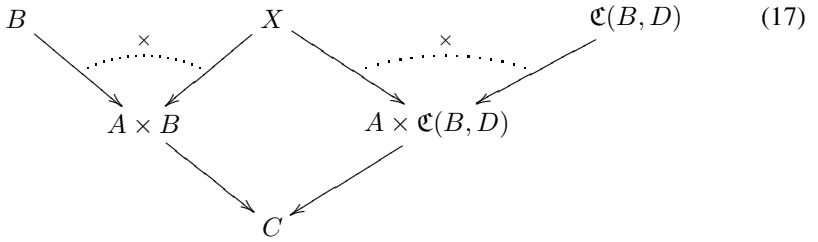
We want to model this source as a pair of relations: $R_1(X, Y)$ and $R_2(P, Q)$, while the sort W should not be exported. Considering the two relations and the functions needed to answer queries on them, we can see that the relation R_1 has two binding patterns: (i, o) and (o, i) , while R_2 has only (o, i) . A query such as “ans(Q) : $-R_1(X, y), R_2(X, Q)$ ” would be rejected by the binding pattern verification system because R_1 produces a set of X values from the query constant y , but R_2 can’t take the X ’s as an input. Mapping the query to a functional diagram, however, produces



which is computable. Therefore, the query can be answered using the model presented in this paper.

Non-binding conditions. Binding patterns are based, as the name suggests, on the idea of binding certain variables in a relation, that is, on the idea of assigning them specific values. Because of these foundations, binding pattern models are ill-equipped to deal with non-binding conditions (that is, essentially, with all conditions except equality and membership in a finite set).

As an example, consider a source with three sorts, A , B , and C , and a function $A \times B \rightarrow C$. in addition, the source has a comparison capability which allows it to compare B with a fourth sort D and return C 's for a specified value of A such that a specified condition $\mathfrak{C}(B, D)$ is verified: $A \times \mathfrak{C}(B, D) \rightarrow C$. the diagram of this source is:



Because the condition $\mathfrak{C}(B, D)$ is non-binding, it doesn't contribute any binding pattern to the relation $R(A, B, C)$ for which the only binding pattern is, therefore, (i, i, o) . A query such as “ans(C) : $\neg R(a, B, C), B < v$ ” where “ $<$ ” is one of the operators allowed for $\mathfrak{C}(B, D)$ is not allowed in the binding pattern model, while it can be executed with the model presented here.

These examples highlight an important general difference between methods, such as binding patterns, that encode the satisfiability of functional constraints in the model, and methods such as ours that verify them when a query is executed: the latter class of methods can take advantage of rewriting opportunities that arise from the specific form of the query, even if they do not apply to a class of queries that can be identified at modeling time.

6 Conclusions

In this paper, we have considered the modeling of data sources for which a relational model doesn't apply, but that can be as a set of functions that, given certain constants and certain conditions, return a set of “corresponding” values. We were interested in modeling these sources as relations, and to find algorithms to translate queries on these relations into combinations of functions provided by the source.

The simplest form of the model that we have presented here is, *mutatis mutandis*, an instance of the problem of finding the closure of a set of functional dependencies and, in this sense, a rather classic one. The framework introduced in that section, however, allowed us to extend the formalism to other problems that either from a modeling point of view (the inclusion of data sorts representing conditions) or algorithmic (the inclusion of joins) go beyond the transitive closure problem.

In these conclusions, we would like to propose a further interpretation of the work presented here, an interpretation that, we believe, is more likely to generate interesting developments. The functions defined for the data source can be seen as atomic statements in a query planning language in which we want to translate our queries, and the function algebra that we have defined constitutes the structural statements of this planning language.

The problem that we have is therefore that of “implementing” queries in a language of fixed structure, but whose primitives change from source to source. In this framework, we can start asking questions such as the optimal structure of the language in order to manage the variability of the statement while still preserving the possibility of easy optimization, or the minimal characteristics of the primitive statements that allow the creation of interesting plans.

Finally, the nature of our method might make static planning (planning done separately from the execution) impossible, because there is no a priori indication of what queries will be feasible and which won’t. It is not clear, at this time, whether static optimal planning is possible for sources with restrictions modeled this way, or if it is necessary to resort to some form of on-the-fly planning as the query is being executed.

These we regard as promising future directions for our work.

References

1. E. Damiani and L. Tanca, “Semantic approaches to structuring and querying web sites,” in *DS-7*, 1997.
2. Z. Liu, F. Li, and W. K. Ng, “Wiccap data model: Mapping physical websites to logical views,” in *Proceedings of ER 2002: 21st International Conference on Conceptual Modeling, Tampere, Finland*, pp. 120–134, October 2002.
3. A. S. da Silva, I. M. Evangelista Filha, A. H. F. Laender, and D. W. Embley, “Representing and querying semistructured web data using nested tables with structural variants,” in *Proceedings of ER 2002: 21st International Conference on Conceptual Modeling, Tampere, Finland*, pp. 135–151, October 2002.
4. V. Zadorozhny, L. Raschid, M.-E. Vidal, T. Urhan, and L. Bright, “Efficient evaluation of queries in a mediator for websources,” in *Proceedings of ACM SIGMOD*, 2002.
5. S. Santini, “Notes on a relational model of functional data sources,” tech. rep., BIRN-CC, University of California, San Diego, 2003. <http://ssantini.ucsd.edu/personal/bibliography/all-by-year/2003/s426long.pdf>.
6. R. Yerneni, C. Li, H. Garcia-Molina, and J. Ullman, “Computing capabilities of mediators,” in *Proceedings of ACM SIGMOD*, pp. 443–454, 1999.
7. Z. Li and H. Chen, “Computing strong/weak bisimulation equivalences and observation congruence for value-passing processes,” *Lecture Notes in Computer Science*, vol. 1579, pp. 300–314, 1999.