A Structure-Based Approach to Querying Semi-Structured Data

Mary Fernandez[1] and Lucian Popa[2] and Dan Suciu[1]

[1] AT&T Labs — Research, Florham Park NJ 07932, USA
[2] University of Pennsylvania, Philadelphia PA 19104, USA

Abstract. Several researchers have considered integrating multiple unstructured, semi-structured, and structured data sources by modeling all sources as edge labeled graphs. Data in this model is self-describing and dynamically typed, and captures both schema and data information. The labels are arbitrary atomic values, such as strings, integers, reals, etc., and the integrated data graph is stored in a unique data repository, as a relation of *edges*. The relation is dynamically typed, i.e. each edge label is tagged with its type.
Although the unique, labeled graph repository is flexible, it looses all static type information, and results in severe efficiency penalties compared to querying structured databases, such as relational or object-oriented databases. In this paper we propose an alternative method of storing and querying semi-structured data, using *storage schemas*, which are closely related to recently introduced graph schemas [BDFS97]. A storage schema splits the graph's edges into several relations, some of which may have labels of known types (such as strings or integers) while others may be still dynamically typed. We show here that all positive queries in UnQL, a query language for semistructured data, can be translated into *conjunctive queries* against the relations in the storage schema. This result may be surprising, because UnQL is a powerful language, featuring regular path expressions, restructuring queries, joins, and unions. We use this technique in order to translate queries on the integrated, semi-structured data into queries on the external sources. In this setting the integrated semi-structured data is not materialized but virtual and the problem is to translate a query against the integrated view, possibly involving regular path expressions and restructuring, into queries which can be answered by the external sources. Here we use again the storage schema in order to split the graph into relations according to their sources. Any positive UnQL query is decomposed based on these relations and translated into queries on the external sources.

1 Introduction

Integration of unstructured and structured data sources is an active area of research. We are particularly interested in integrating the unstructured graphs of HTML pages with one or several structured (relational or object-oriented) databases, into a single web site. We do the integration by *modeling* all data sources as edge-labeled graphs. Such graphs are an appropriate model for integrating various types of data sources, because they can model unstructured data that have no fixed schema, semi-structured data whose schema imposes only weak constraints on the data, and structured data that have fixed schema. Both schema information and data are integrated into a single graph [PGMW95], which is self-describing and dynamically typed. The graphs' labels can be arbitrary atomic types, such as strings, integers, reals, etc., and the graph is stored in a common data repository in which each label is tagged with its type. Several query languages have been proposed for querying semi-structured data modeled as edge-labeled graphs: for example Lorel [QRS+95], UnQL [BDHS96a]. All of them allow some kind of recursion, typically in the form of regular path expressions, which serves two purposes: either to browse through a structured data source whose structure is unknown to the user, or to query a truly unstructured data source, such as the graph of a collection of HTML pages, which requires traversal of arbitrarily long paths.

Although we want to model various data sources as graphs, we do *not* want to materialize them as graphs, because materializing is prohibitively expensive and raises the problems of maintaining consistency between the original and materialized data. Moreover, we would like to have the external sources answer queries directly, by translating queries expressed against the integrated graph data into queries on the external sources. In general these queries are recursive, because they contain regular path expressions: when the sources have known structure, then recursive queries are typically unfolded into non-recursive ones. But when the sources are truly unstructured, then the recursion cannot be eliminated. For example, in our application of integrating HTML pages with relational databases we would like to translate such a query into a recursive query on the portion of the graph described by the HTML pages and into relational queries on the relational databases.

Representing a graph as a single relation, however, complicates the translation, because the relation has all type and source information stored dynamically, instead of statically. In this paper we propose a new method of storing and querying semi-structured data based on *storage schemas*, a concept related to recently introduced graph schemas [BDFS97]. A storage schema divides the semi-structured graph data into several relations, some of which contain statically typed, monomorphic edges and some of which remain dynamically typed. When the semi-structured data is virtual, then these relations are virtual as well, i.e. they are queries on the external sources. We show that this method is effective by proving that it can support the evaluation of queries in a semi-structured query language. For that we chose UnQL [BDHS96a], a query language powerful enough to support regular path expressions and restructurings. We show that any positive UnQL query Q can be evaluated as follows. Given a storage schema for the input graph data, we compute a new storage schema describing how Q's answer is stored. Each relation in the new storage schema is a conjunctive query on some of the relations of the input data graph. Thus, when the input relations are statically typed, so is the resulting storage schema and so are all conjunctive queries. Moreover, when the input storage schema is virtual, then the output storage schema contains queries which reach directly into the external sources, typically in the form of relational algebra (RA) queries. For relational external sources we go one step further and show how to extract the result from the output storage schema as a single SQL expression, in the process eliminating the recursion from the original query.

We begin in Section 2 with an extended example that illustrates the main ideas. In Section 3 we review the data model, graph schemas, and query language. We introduce storage schemas in Section 4, and state our main translation result in Section 5. We briefly describe the main ideas behind the translation in Section 6, and show two applications in Sections 7 and 8.

2 An Example

Consider the following UnQL query, which searches for an edge y with a label that contains the string *"Security"* and returns the surrounding structure:

$$\begin{aligned} \mathit{Query}\ Q1 : \textsf{select}\ & \{x \Rightarrow \{y \Rightarrow t\}\} \\ \textsf{where}\ & * \Rightarrow x \Rightarrow y \Rightarrow t\ \textsf{in}\ DB, \\ & y\ \textsf{matches}\ "\mathit{Security}" \end{aligned}$$

For example, on the graph database in Figure 1, the query returns the graph in Figure 2(a). UnQL actually produces the more complex graph in Figure 2(b), which includes *epsilon (ε) edges*. Intuitively, ε edges represent the edges in the source graph that are not part of the result and are eliminated from the result graph. Epsilon edges are important in the implementation of our translation technique, described later.

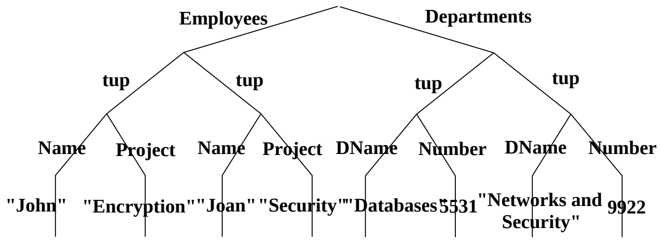


Fig. 1. Example graph database, DB_1.

Labeled graphs modeling semi-structured data are stored in two relations: a ternary relation, $E(\mathit{Oid}, \mathit{Label}, \mathit{Oid})$, storing the edges, and a unary one, $\mathit{Root}(\mathit{Oid})$, storing the root [PGMW95,AV97]. *Oid* is a graph node's unique identifier and *Label* is an edge label. Since atomic values can be of any type, e.g., *Int*, *String*, *Bool*, etc., *Label* must be their disjoint union, i.e., $\mathit{Label} = \mathit{Int} \cup \mathit{String} \cup \mathit{Bool} \cup \ldots$

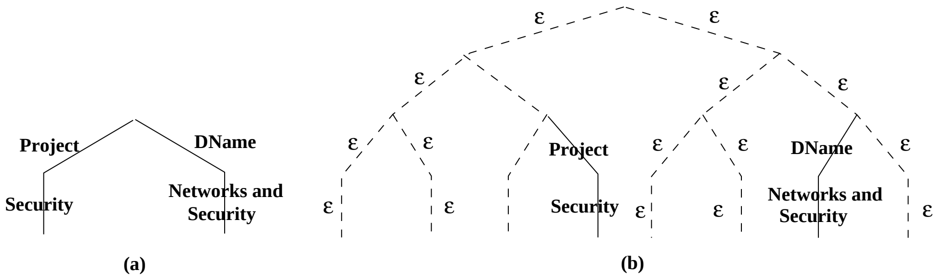


Fig. 2. Result of applying query $Q1$ to graph database, DB_1.

Given this untyped, relational representation, we could translate our example query into the Datalog program:

$$
\begin{aligned}
&C(r) && :-\ \mathit{Root}(r)\\
&C(n') && :-\ C(n), E(n, \_, n')\\
&\mathit{Answ}(r, x, n') && :-\ \mathit{Root}(r), C(n), E(n, x, n'), E(n', y, n''),\\
& && \quad \textsf{matches}\ (y, "\mathit{Security}")\\
&\mathit{Answ}(n', y, n'') && :-\ C(n), E(n, x, n'), E(n', y, n''), \textsf{matches}\ (y, "\mathit{Security}")
\end{aligned}
$$

If we know, as we do in our example, that the data is a graph with bounded depth, we could unfold the Datalog program into the equivalent non-recursive program:

$$
\begin{aligned}
&\mathit{Answ}(r, x, n') && :-\ \mathit{Root}(r), && // \text{ Unfolding 1}\\
& && \quad E(r, x, n'), E(n', y, n''), \textsf{matches}\ (y, "\mathit{Security}")\\
&\mathit{Answ}(n', y, n'') && :-\ \mathit{Root}(r),\\
& && \quad E(r, x, n'), E(n', y, n''), \textsf{matches}\ (y, "\mathit{Security}")\\
&\mathit{Answ}(r, x, n') && :-\ \mathit{Root}(r), E(r, \_, n) && // \text{ Unfolding 2}\\
& && \quad E(n, x, n'), E(n', y, n''), \textsf{matches}\ (y, "\mathit{Security}")\\
&\mathit{Answ}(n', y, n'') && :-\ \mathit{Root}(r), E(r, \_, n)\\
& && \quad E(n, x, n'), E(n', y, n''), \textsf{matches}\ (y, "\mathit{Security}")\\
&\mathit{Answ}(r, x, n') && :-\ \mathit{Root}(r), E(r, \_, n_1), E(n_1, \_, n), && // \text{ Unfolding 3}\\
& && \quad E(n, x, n'), E(n', y, n''), \textsf{matches}\ (y, "\mathit{Security}")\\
&\mathit{Answ}(r, y, n'') && :-\ \mathit{Root}(r), E(r, \_, n_1), E(n_1, \_, n),\\
& && \quad E(n, x, n'), E(n', y, n''), \textsf{matches}\ (y, "\mathit{Security}")
\end{aligned}
$$

For our example graph data, the first two unfoldings are vacuous, but without explicit knowledge on the graph's structure, they cannot be eliminated. Like the original recursive query, the non-recursive query does not preserve any type

Employees

Name	Project
John	Encryption
Joan	Security

Departments

DName	Number
Databases	5531
Networks and Security	9922

Fig. 3. Relational database.

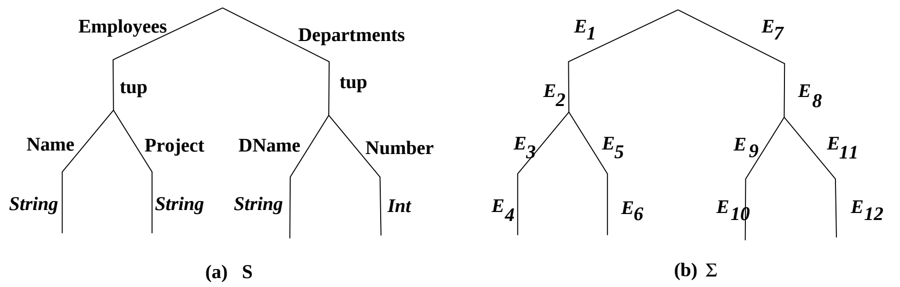


Fig. 4. (a) Graph schema, S, and (b) storage schema, Σ, for relational database in Figure 3.

information; for example, we cannot determine from the $Answ$ rules in the third-level unfolding that x is always a string and that y can be a string or an integer.

We propose a technique for representing graph data that preserves the type and structure information associated with the source data. To illustrate, assume the external data source for Figure 1 is the relational database in Figure 3. Then we can describe the graph's structure using a *graph schema* [BDFS97]: for our example the graph schema S is in Figure 4 (a). It specifies that the data graph has two first-level edges labeled "Employees" and "Departments". Each second-level edge in the data graph is labeled "tup". Then, the third level edges are labeled with the strings "Name", "Project", "DName", or "Number" respectively. The first three are followed by edges labeled with strings, while the last is followed by edges labeled with integers. The graph schema is only intended to say what labels are allowed to occur in the graph database. It cannot prevent labels to be missing, nor does it say anything about unique v.s. multiple occurrences, see [BDFS97].

We push the concept of graph schema one step further and define a *storage schema*, to describe a specialized storage of the data graph. In our example, the corresponding storage schema is in Figure 4(b). Each of the 12 relations $E_1, \ldots, E_{12}$ stores a fragment of the original edge relation E, namely the graph fragment that contains those edges that "populate" the corresponding schema edge. While $E = E_1 \cup \ldots \cup E_{12}$, notice that the new edge relations are strongly typed, e.g., $E_1, \ldots, E_{11}$ have the type $(Oid, String, Oid)$, and E_{12} has the type (Oid, Int, Oid). The translation from the relational database $Employees(Name, Project)$, $Departments(DName, Number)$ to the storage schema is handled by a wrapper which computes the 12 relations E_i, each of which is a conjunctive query on *Employees* and *Departments*, see Section 7.

We show now how query $Q1$ can be evaluated in terms of the storage schema. A key observation is that the answer of a query can be also stored according to some storage schema. For example, Figure 5(a) depicts the graph schema for the result of applying $Q1$ to graphs that conform to the schema in Figure 4(a). Note that the query's result in Figure 2(b) conforms to the output schema in Figure 5(a). Figure 5(b) depicts the result's storage schema. Each of the 20 relations, Q_i, can be expressed as a conjunctive query over the input relations $E_1, \ldots, E_{12}$. Most of the queries depend on a single relation, some depend on two:

$$\begin{array}{ll}
Q_3(n,x,n') & :- \; E_3(n,x,n'), E_4(n',y,n''), \textsf{matches}\ (y, "Security") \\
Q_5(n,x,n') & :- \; E_5(n,x,n'), E_6(n',y,n''), \textsf{matches}\ (y, "Security") \\
Q_9(n,x,n') & :- \; E_9(n,x,n'), E_{10}(n',y,n''), \textsf{matches}\ (y, "Security") \\
Q_{11} & := \; \emptyset \\
Q_i(n',y,n'') & :- \; E_i(n',y,n''), i = 4,6,10,12 \\
Q_i(n,\varepsilon,n') & :- \; E_i(n,_,n'), i = 1,2,7,8 \\
Q'_i(n,\varepsilon,n') & :- \; E_i(n,_,n'), i = 3,4,5,6,9,10,11,12
\end{array}$$

The type information preserved in the storage schema allows us to optimize

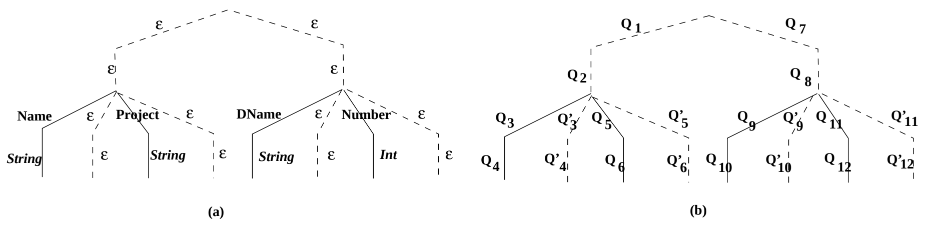


Fig. 5. (a) Graph schema, $S' = Q1(S)$, for result of query $Q1$. (b) Storage schema, $\Sigma' = \bar{Q}1(\Sigma)$, for query result.

queries that we could not optimize in the single edge relation representation. For example, we can simplify the query

$$Q_{11}(n,x,n') : -E_{11}(n,x,n'), E_{12}(n',y,n''), \textsf{matches}\ (y, "Security")$$

to $Q_{11} := \emptyset$, because we know that the y attribute in E_{12} is always an integer value and can never match a string value. Finally, we can push the queries Q_i to the external source, by composing the Q_i queries with the conjunctive queries E_i, which are defined in terms of the original *Employees* and *Departments* relations. Note that by separating a graph's edges according to a storage schema we can use the same oid's for distinct nodes in the resulting graph. For example the edges populating Q_4 and Q'_4 use the same oid's for their end nodes, but these oid's denote different nodes.

The main application of our technique is translating queries on an integrated, semi-structured information system into queries on the external sources. Other systems, such as Tsimmis [PGMU95,PGGMU95,PAGM96], address this problem. Tsimmis uses a datalog-like language, called MSL, to integrate data from heterogeneous information sources into a graph-like, semi-structured data model. A query against the integrated information source is then composed with the MSL query. When both queries are non-recursive, the resulting query can be efficiently "pushed to the sources". But their technique does not work with recursion. We address the same problem here, but allow recursion, which is indispensable in our application, because one of the data sources (the graph of HTML pages) is unstructured. For example, consider the data in Figure 6 (a), in which some HTML pages are integrated with the *Employees* relation from our previous example into a unique Web site. The bold edges denote the fragment of the data that is fetched from the relational database, other edges denote HTML hyper-links. The idea is that every employee page will have a link to a piece of information from *Employees* relation: for example Joan's page points to the *Employees* tuple with $Name =$ "$Joan''$. We shall explain in Section 8 how this integration is achieved. Suppose that we are evaluating the query $Q1$ above on this integrated data: "Security" edges may be found both in the HTML graph and the relational part of the data. We want to translate $Q1$ into a recursive query on the HTML graph and a relational query on the *Employee* relation. We show how to do this using storage schemas. First, the graph schema for this example is in Figure 6 (b) and the storage schema in (c). The latter says that the data consists of (1) a graph relation $E1$ holding all the HTML links, and (2) five virtual relations $E2, E3, \ldots, E6$ defined in terms of the *Employee* relation. $E3, \ldots, E6$ are conjunctive queries on *Employee*, while $E2$ does the matching between HTML links labeled with names and their corresponding entry in *Employees*. Applying our translation technique to $Q1$ and Σ_2 results in a storage schema[1] whose associated graph schema is S'_2 in Figure 6(d). We shall see in Section 8 how this storage schema helps us generating some relational queries on *Employees* and a recursive query on the HTML graph (corresponding to the ε loop on the top of Figure 6(d)).

3 Background

Data Model. We use the data model described in [BDHS96a,BDHS96b] for unstructured data. A *database* is a *rooted, edge-labeled graph*, with labels from an infinite universe $Label = Int \cup String \cup \ldots$ Only the part of the graph accessible from the root matters, but sometimes we store also unaccessible nodes and edges because removing them can be expensive. Each graph can be viewed as a *set* of label-graph pairs: $\{a_1 \Rightarrow t_1, \ldots, a_n \Rightarrow t_n\}$, where $a_1, \ldots, a_n \in Label$ and $t_1, \ldots, t_n$ are other graphs. For example, the databases in Figure 7 (a) and (b)

[1] The storage schema is depicted in Figure 13(a).

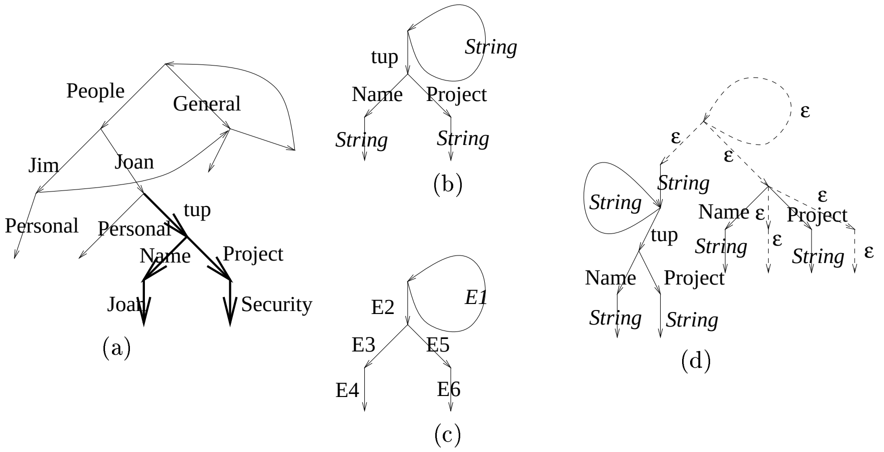


Fig. 6. (a) Integrated HTML and relational data graph, DB_2. (b) Database DB_2's graph schema, S_2. (c) Graph schema S_2's storage schema Σ_2. (d) Graph schema $S_2' = \bar{Q}1(S_2)$.

can be written as: $t = \{a \Rightarrow \{c, d \Rightarrow \{e\}\}, b \Rightarrow \{f\}\}$ and $t' = \{a \Rightarrow \{b\} \cup T\}$ where $T = \{c \Rightarrow T\}$ respectively. Since graphs are sets, we can define the *union* of two graphs $t \cup t'$ and the *empty* graph $\{\}$, which has one node and no edges. For t, t' above, $t \cup t' = \{a \Rightarrow \{c, d \Rightarrow \{e\}\}, b \Rightarrow \{f\}, a \Rightarrow \{b \cup T\}\}$ in Figure 7 (c). We shall use the special symbol ε to label "silent" or "empty" edges. An edge $x \stackrel{\varepsilon}{\to} y$ means that every label visible from node y is also visible from node x. That is, we may remove the ε edge, if for every edge $y \stackrel{a}{\to} z$ we introduce a new edge $x \stackrel{a}{\to} z$. The silent edges are just a convenient, concise notation. For example the union of the graphs in Figure 7 (a) and (b) can be represented as in (d). A formal treatment of how to remove ε-edges uses bisimulation [BDHS96a].

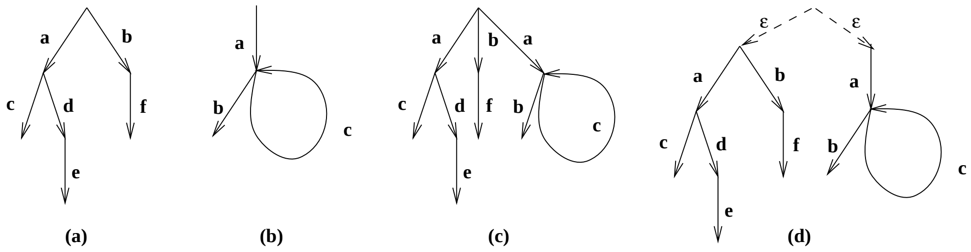


Fig. 7. Examples of rooted graphs.

Graph Schemas. In recent work we introduced *graph schemas* [BDFS97] for the purpose of query optimization. A graph schema is a graph with n nodes $s_1, \ldots, s_n$ called *states*, a designated *root* state s_1, and edges labeled with unary predicates over *Label*. Figure 4(a) represents a schema with 13 states. An edge with the constant label "Employees" denotes the predicate $l =$"Employees"; an edge with predicate label *String* denotes the predicate $isString(l)$.

Graph schemas recapture some of the structural and type information lost by unifying schema information and data in a single graph. Graph schemas, however, are more like *data guides* [Abi97] than like traditional database schemas, because they do not fully describe a graph's structure. For example, Figure 6(b) does not specify the structure of the graph above the "tup" edges.

Query Language. UnQL is a query language for querying and restructuring data modeled as rooted graphs. It is compositional and supports regular path expressions. A complete description appears elsewhere [BDHS96a]. UnQL is equivalent to UnCAL, a calculus in which the salient construct is a simple form of recursion on trees called *gext*: UnQL's regular path expressions and its more complex restructuring constructs can all be translated into this form of tree recursion. We will use UnCAL rather than UnQL to describe the evaluation technique based on storage schemas. UnCAL is revised in Appendix A. Here we briefly review it's salient construct, its recursion on trees.

UnCAL's *gext* construct allows us to define k mutually recursive functions, $g_1, \ldots, g_k$, as in Figure 8. Each function g_i returns the empty graph on the empty graph, $\{\}$, and commutes with union. On a singleton graph $\{l \Rightarrow t\}$, where l is an edge and t is a graph, it may call recursively any of the functions $g_1(t), \ldots, g_k(t)$ and combine their results into an expression F_i, which may also depend on l and t. Thus the functions are defined as if they were applied to trees only, and not to graphs with cycles. The important observation in [BDHS96a,BDHS96b] is that by imposing minor restrictions on the form of the expressions F_i we can guarantee that they are well defined on graphs with cycles too[2]. The expressions F_i may have other recursive functions defined inside. For example query $Q1$ of Section 2 can be expressed as a recursive function g, having another recursive function inside, see Figure 9.

$$\begin{array}{rlcrl}
g_1(\{\}) & \stackrel{\text{def}}{=} \{\} & \ldots & g_k(\{\}) & \stackrel{\text{def}}{=} \{\} \\
g_1(\{l \Rightarrow t\}) & \stackrel{\text{def}}{=} F_1(l, t, g_1(t), \ldots, g_k(t)) & \ldots & g_k(\{l \Rightarrow t\}) & \stackrel{\text{def}}{=} F_k(l, t, g_1(t), \ldots, g_k(t)) \\
g_1(t \cup t') & \stackrel{\text{def}}{=} g_1(t) \cup g_1(t') & \ldots & g_k(t \cup t') & \stackrel{\text{def}}{=} g_k(t) \cup g_k(t')
\end{array}$$

Fig. 8. Defining k mutually recursive functions.

[2] We will use of this fact in the translation, Appendix A: there we define functions $f_1, \ldots, f_k$ by recursion on tree storage schemas (Figure 15), and argue that they are still well defined on storage schemas with cycles.

$$
\begin{aligned}
g(\{\}) &\stackrel{\text{def}}{=} \{\} \\
g(\{x \Rightarrow t\}) &\stackrel{\text{def}}{=} \mathsf{let}\ g'(\{\}) \stackrel{\text{def}}{=} \{\} \\
&\qquad g'(\{y \Rightarrow t'\}) \stackrel{\text{def}}{=} \mathsf{if}\ (y\ \mathsf{matches}\ \text{``}Security\text{''})\ \mathsf{then}\ \{x \Rightarrow \{y \Rightarrow t'\}\} \\
&\qquad\qquad \mathsf{else}\ \{\} \\
&\qquad g'(t'_1 \cup t'_2) \stackrel{\text{def}}{=} g'(t'_1) \cup g'(t'_2) \\
&\quad \mathsf{in}\ g(t) \cup g'(t) \\
g(t_1 \cup t_2) &\stackrel{\text{def}}{=} g(t_1) \cup g(t_2)
\end{aligned}
$$

Fig. 9. Query $Q1$ of Section 2 expressed as a recursive function.

Conjunctive queries Conjunctive queries are extensively covered in textbooks, see for example [AHV95]. For the purpose of our paper, we need conjunctive queries in a different formalism, namely as a certain fragment of the comprehension syntax [BLS+94,Won94]. We call this formalism CONJ. It has two kinds of expressions: scalar and set expressions. Scalar expressions are $S ::= \langle S_1, \ldots, S_n\rangle \mid \pi_i(S) \mid z \mid a$, where $\langle \ldots \rangle$ constructs a tuple, $\pi_1, \pi_2, \ldots$ denote projections, z is a scalar variable, and $a \in Int \cup String \cup \ldots$ is a constant. The set expressions denote flat sets (we don't have nested sets) and are $E ::= \{S \mid C_1, \ldots, C_n\} \mid R_1 \mid \ldots \mid R_m$. Here $R_1, \ldots, R_m$ are relation names. The core of the comprehension syntax is the comprehension, $\{S \mid C_1, \ldots, C_n\}$: here each of C_i is either a *generator*, i.e., $z \in E$ with z a variable and E a set expression, or a *condition*, i.e., $S = S'$ or $S \neq S'$. For example, the expression $\{\langle \pi_1(z), \pi_2(z')\rangle \mid z \in R_1, z' \in R_2, \pi_2(z) = \pi_1(z')\}$ computes the join of R_1 and R_2. When convenient, we use pattern matching for shorthand, e.g., the join query can be written as $\{\langle x, y\rangle \mid \langle x, v\rangle \in R, \langle w, y\rangle \in R', v = w\}$.

Conjunctive queries are strongly typed. The scalar types are given by:

$$
\begin{aligned}
b &::= Int \mid String \mid \ldots \mid \varepsilon \qquad \text{(base types)} \\
s &::= b \mid (s_1 \times \ldots \times s_n)
\end{aligned}
$$

We allow a special base type ε, with a single value, ε. In this paper, we distinguish between $s_1 \times (s_2 \times s_3)$ and $s_1 \times s_2 \times s_3$. We call *Unit* the empty cartesian product ($s_1 \times \ldots \times s_n$ with $n = 0$); its only value is the empty tuple, $\langle\rangle$. A *set type* has the form $\{s\}$.

4 Storage Schemas

A *storage schema* is a rooted graph whose edges are labeled with conjunctive queries over some fixed, relational schema $R_1, \ldots, R_m$. We call its nodes *states*. Since storage schemas look like graph data, we use the same notation for them, e.g. $\{E \Rightarrow \{E'\}, E''\}$ etc. When we write the CONJ queries explicitly, we delimit them to help distinguish between the two languages. For example, $\{\boxed{\{\langle x,y,y\rangle \mid \langle x,y\rangle \in R\}} \Rightarrow \{\boxed{\{\langle z,z,z\rangle \mid z \in R'\}}\}\}$ denotes a storage schema with 3 states and two edges.

We require storage schemas to be well typed. More precisely, we say a storage schema Σ is well typed if there exists an assignment of set types to its states such that if some state has type $\{s\}$, then all incoming edges are labeled with CONJ expressions of type $\{s' \times b' \times s\}$, for some types s', b', and all outgoing edges have type $\{s \times b'' \times s''\}$, for some types b'', s''.

Storage schemas extend the graph schemas. The latter assign predicates to the edges, saying what types the edges in the data graph can be. The storage schema goes one step further and associates a ternary relation for each schema edge, in effect defining the set of all data edges populating the schema edge. The type can be recaptured from the type of the ternary relation: when an edge in the storage schema is labeled with a CONJ expression of type $\{s \times b \times s'\}$, then the predicate in the corresponding graph schema is $P(l) \stackrel{\text{def}}{=} l \in b$.

A storage schema Σ is intended to be a recipe for storing a data graph, namely by storing one ternary relation for each edge in Σ. Given an instance of the relational database $R_1, \ldots, R_m$, we can recover the data graph as follows. Let $s_1, \ldots, s_n$ be Σ's states, and let E_{ij} be the CONJ expression labeling the edge $s_i \to s_j$ (or $E_{ij} = \emptyset$ if there is no edge). Then the graph database DB has as edges the disjoint union of all E_{ij}'s: $E \stackrel{\text{def}}{=} \bigsqcup E_{ij}$. We use certain terminology to relate DB to the storage schema Σ. We say that an edge $e \in E_{ij}$ *populates* or *belongs to* the edge $s_i \to s_j$ in Σ. Similarly we say that a node $n \in \Pi_1(E_{ij})$ or $n \in \Pi_3(E_{ki})$ *belongs to* the state s_i. We explicitly allow the same value to denote different nodes if it belongs to different schema states. For example, assume that the schema states s_4 and s_6 have type *String* and that, say "abc" populates[3] both s_4 and s_6, then DB will have two nodes corresponding to two distinct copies of the string "abc": one belonging to s_4 the other to s_6. Finally DB's root is, by definition, the node $\langle\rangle$. That implies that Σ's root state, s_1, has type *Unit*. This can be always achieved, e.g. by adding an extra state and a ε edge in Σ.

We explicitly allow storage schemas to be populated by ε edges. That is, certain edges in Σ are labeled with CONJ expressions E_{ij} of type $\{s \times \varepsilon \times s'\}$, where ε is the atomic type containing the single value ε. For example the 12 edges Q_1, Q_2, Q'_3, Q'_4, Q'_5, Q'_6, Q_7, Q_8, Q'_9, Q'_{10}, Q'_{11}, Q'_{12} in the storage schema Σ of Figure 5 (b) are populated only by ε edges. If these edges form no cycle in the storage schema, then all ε-edges can be eliminated by a relational algebra query, by essentially computing joins. For example, for Σ above, the edges of the ε-free data graph are given by[4] $E = \Pi_{167}(Q_1 \bowtie Q_2 \bowtie Q_3) \cup \Pi_{167}(Q_1 \bowtie Q_2 \bowtie$

[3] That is, "abc" belongs both to $\bigcup_{j=1,n} \Pi_1(E_{4i}) \cup \bigcup_{k=1,n} \Pi_3(E_{k4})$ and to $\bigcup_{j=1,n} \Pi_1(E_{6i}) \cup \bigcup_{k=1,n} \Pi_3(E_{k6})$.

[4] To keep the example simple, we assume that the states in Σ are populated by disjoint sets of nodes.

$Q_5) \cup \Pi_{167}(Q_7 \bowtie Q_8 \bowtie Q_9) \cup \Pi_{167}(Q_7 \bowtie Q_8 \bowtie Q_5 11) \cup Q_4 \cup Q_6 \cup Q_{10} \cup Q_{12}$: this constructs the data graph in Figure 2 (a). However when Σ has cycles of ε-edges, then we need to compute transitive closure in order to produce a ε-free data graph.

5 Main Result

Theorem 1. *Let Q be a positive UnQL query and Σ a storage schema. Then there exists a storage schema Σ' over the same relational schema as Σ, which can be effectively computed from Σ and Q, such that: for any instance of the relational schema, if DB and DB' are the graph databases denoted by Σ and Σ' respectively, then*[5] *$Q(DB) = DB'$. We denote Σ' with $\bar{Q}(\Sigma)$.*

The theorem says that we can evaluate the UnQL query Q by essentially evaluating a collection of conjunctive queries. Namely first compute the resulting storage schema Σ', based on Σ and Q. Then evaluate each conjunctive query on the edges of Σ'. The result denotes a graph which is precisely $Q(DB)$. So far there was no need to compute any recursive queries. However the resulting graph, $Q(DB)$, typically contains lots of ε edges. If the ε-edges form no cycle in Σ', then we can still eliminate them, as described in Section 4. Otherwise we need to compute a recursive query.

In Section 6 we describe the main ideas underlying Theorem 1. Appendix A describes the complete translation. The formal proof is deferred to the full version of the paper.

6 Translating Positive UnQL Queries into Conjunctive Queries

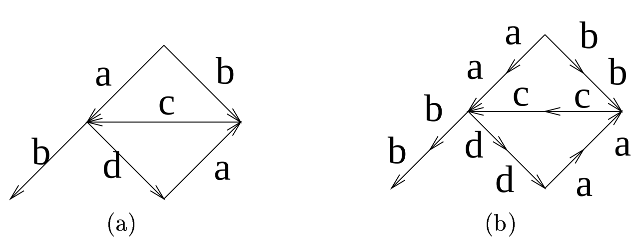


Fig. 10. (a) A graph database, DB. (b) $gext_Z(\lambda(l,t).(Z := \{l \Rightarrow \{l \Rightarrow Z\}\}))(DB)$

[5] Equality $Q(DB) = DB'$ should be read here as bisimulation [BDHS96a].

Translating recursive functions. Consider the UnCAL query $Q2(DB) = g_1(DB)$ where g_1 is:

$$g_1(\{\}) \stackrel{\text{def}}{=} \{\}$$
$$g_1(\{l \Rightarrow t\}) \stackrel{\text{def}}{=} \{l \Rightarrow \{l \Rightarrow g_1(t)\}\}$$
$$g_1(t_1 \cup t_2) \stackrel{\text{def}}{=} g_1(t_1) \cup g_1(t_2)$$

Formally it is written in UnCAL as $Q2(DB) = (gext_Z(\lambda(l,t).(Z := \{l \Rightarrow \{l \Rightarrow Z\}\})))(DB)$. It copies the entire graph DB, doubling each edge (see Figure 10). Assume that the data is stored according to the storage schema in Figure 11 (a) and that the relation $E2$ is populated by the six edges in Figure 11(b), i.e., $E2 = \{\langle x,a,u\rangle, \langle x,b,v\rangle, \langle v,c,w\rangle, \langle z,d,w\rangle, \langle z,e,p\rangle, \langle z,f,q\rangle\}$. After applying $Q2$ to the input graph we expect each edge to be doubled (Figure 12(a)). This graph fragment has six new intermediate nodes, for which we have to "create" oid's in CONJ. The new nodes correspond to the six edges populating $E2$, so we name the new nodes after these edges: each new node is a triple of the form $\langle n,l,n'\rangle \in E2$. Now we have non-uniformly named nodes in this graph fragment: some are atomic values (like $x, y, z, u, \ldots$) while others are triples[6] (like $\langle x,a,u\rangle, \langle x,b,v\rangle, \ldots$). For uniformity, we replace this graph with another one, depicted in Figure 12(b), which adds some ε-edges. Here the "internal" nodes are uniformly named using triples, while "input-output" nodes have names from the old graph. We describe these intermediate results by the storage schema fragment in Figure 12(c), where $E2_{\text{in}}$ and $E2_{\text{out}}$ store the input and output ε-edges, and $E2'$ and $E2''$ store the internal edges. The expressions for these four relations are:

$$E2_{\text{in}} = \{\langle n, \varepsilon, \langle n,l,n'\rangle\rangle \mid \langle n,l,n'\rangle \in E2\}$$
$$E2_{\text{out}} = \{\langle\langle n,l,n'\rangle, \varepsilon, n'\rangle \mid \langle n,l,n'\rangle \in E2\}$$
$$E2' = E2'' = \{\langle\langle n,l,n'\rangle, l, \langle n,l,n'\rangle\rangle \mid \langle n,l,n'\rangle \in E2\}$$

Putting everything together, we derive the storage schema $\Sigma_3' = \bar{Q}2(\Sigma_3)$ in Figure 12(d).

Other constructs. We translate a union expression $e_1 \cup e_2$ by first translating e_1, e_2 to obtain the storage schemas Σ_1, Σ_2. The result storage schema will be $\{E \Rightarrow \Sigma_1, E \Rightarrow \Sigma_2\}$, where E is the following CONJ expression: $\{\langle\langle\rangle, \varepsilon, \langle\rangle\rangle\}$. Note that the resulting root in the data graph is $\langle\rangle$, and that this construct assumes that the roots of the data graphs denoted by Σ_1, Σ_2 are also $\langle\rangle$. A conditional $\mathsf{if}\ l = l'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ is translated into: $\{E_1 \Rightarrow \Sigma_1, E_2 \Rightarrow \Sigma_2\}$, where E_1, E_2 are CONJ expressions denoting either ε-edges or the empty set. Namely: $E_1 = \{\langle\langle\rangle, \varepsilon, \langle\rangle\rangle \mid l = l'\}$, while $E_2 = \{\langle\langle\rangle, \varepsilon, \langle\rangle\rangle \mid l \neq l'\}$.

Implementation. We have implemented an UnCAL interpreter in ML and are currently implementing a translator. The core of the translator is the translation of *gext*: here the translator is described in terms of k recursive functions

[6] One can think of these triples as semantic oid's.

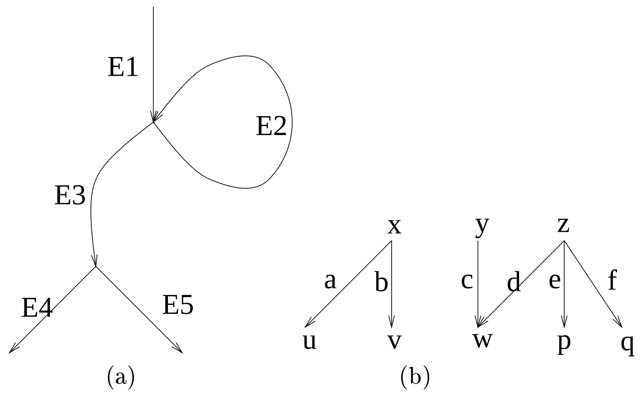


Fig. 11. (a) A storage schema, Σ_3, and (b) six edges populating $E2$.

$f_1, \ldots, f_k$, Figure 15. These functions have to traverse a storage schema Σ, which in general can be cyclic. We used here the crucial observation [BDHS96a,BDHS96b] that such recursive functions are well defined on cyclic graphs too. In the implementation we went one step further and realized that these recursive functions are an instance of *gext*, when given the right context. Namely let us denote the set of all data graphs by $Graph(Label)$, emphasizing that the graphs' edges are labeled with elements from $Label$. Then the set of storage schemas is $Graph(\text{CONJ})$, i.e., they are data graphs, but whose edges are labeled with expressions in CONJ. Similarly, UnCAL($Label$) denotes the set of UnCAL queries $Q : Graph(Label) \to Graph(Label)$. Now it makes sense to talk about the "query language" UnCAL(CONJ): a query in this language, $\bar{Q}$ maps an input storage schema Σ into some output storage schema $\Sigma' = \bar{Q}(\Sigma)$, i.e. $\bar{Q} : Graph(\text{CONJ}) \to Graph(\text{CONJ})$. The language UnCAL(CONJ) has the same primitives as UnCAL($Label$), including a *gext* construct which allows us to define the functions $f_1, \ldots, f_k$ from Figure 15. In addition it has a set of external functions operating on the "labels", which are now CONJ expressions, which allow us to construct more complex CONJ expressions from simpler ones. In our implementation, we translate $Q \in \text{UnCAL}(Label)$ into $\bar{Q} \in \text{UnCAL}(\text{CONJ})$, such that the following diagram commutes:

$$\begin{tikzcd} Graph(Label) \arrow[r, "Q"] & Graph(Label) \\ Graph(\text{CONJ}) \arrow[u, "Denotes"] \arrow[r, "\bar{Q}"] & Graph(\text{CONJ}) \arrow[u, "Denotes"'] \end{tikzcd} \qquad (1)$$

Optimizations. The CONJ expressions derived during the translation are highly redundant, but can be simplified dramatically by using the optimization rules

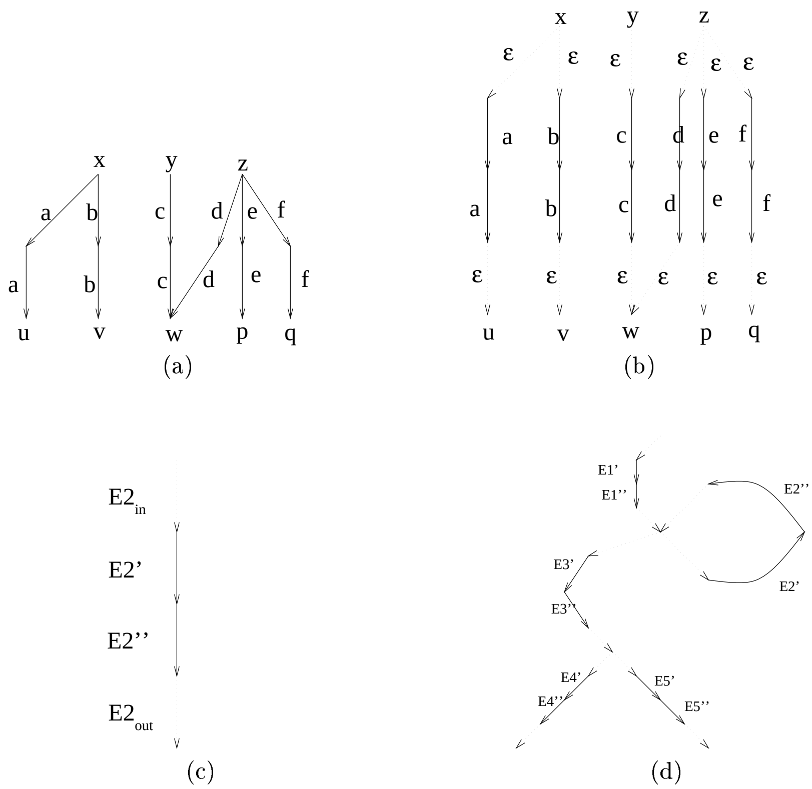


Fig. 12. (a) and (b) Result of "double-edge" query. (c) Result's storage schema for edge E_2. (d) Complete storage schema, $\Sigma_3' = \bar{Q}2(\Sigma_3)$.

in [Won94]. The most effective rule is "vertical loop fusion", which eliminates intermediate results. Also, many of the edge expressions have a constant label value in the second position, e.g., the E_3 in Figure 4(b) is of the form $\{\langle n, "Name", n' \rangle \mid \ldots\}$. When these expressions occur in conditionals generated by UnCAL expressions, they can often be reduced to empty sets.

We identified another rule, which we call *join elimination*, that can be applied when the result expression involves only one of the input relations; this is a novel rule, not mentioned among the optimization rules in [Won94]. The rule is:

$$\{x_1 \mid x_1 \in e, x_2 \in e, x_1 = x_2\} \rightarrow e$$

7 Applying UnQL queries to relational databases

There are several reasons why we would like to translate UnQL queries on relational databases into SQL. First, UnQL can easily express *browsing* queries, like "find all records containing the string 'Security' in the database". Our translation uses the source's relational schema to generate several SQL queries that can do the browsing. Second, this translation is a starting point for the more ambitious goal of pushing queries to multiple external sources that are components of an integrated, semi-structured database. We describe that problem in the next section. Third, the translation offers the base of an alternative proof that UnQL is a conservative extension of the relational algebra [BDHS96a].

In our translation of UnQL queries into SQL, we consider a relational database encoded as a tree data-graph (see Figure 1). Given an UnQL query Q, we search for an "equivalent" SQL query Q'. In UnQL, we can write queries returning nested sets, therefore there won't always find an equivalent SQL query. Even when the result is not nested, it may return a heterogeneous set, for example query $Q1$ in Section 2. In this case, although there is no equivalent SQL query, Q is equivalent to the union of several SQL queries, returning sets of different types.

Tree Schemas. Any relational schema can be represented as a tree with depth four. The corresponding storage schema is obtained as follows. We will refer to Figure 4(b), where the relational schema is $Employee : \{String \times String\}$, $Departments : \{String \times Int\}$:

1. The root state and the states at the first level are populated with the empty tuple $\langle\rangle$.

$$E_1 \stackrel{\text{def}}{=} \{\langle\langle\rangle, \text{"}Employees\text{"}, \langle\rangle\rangle\}$$
$$E_7 \stackrel{\text{def}}{=} \{\langle\langle\rangle, \text{"}Departments\text{"}, \langle\rangle\rangle\}$$

2. The nodes at the second level are populated with records from the corresponding relation. The CONJ expressions are fan-out subgraphs. In our example:

$$E_2 \stackrel{\text{def}}{=} \{\langle\langle\rangle, \text{"}tup\text{"}, z\rangle \mid z \in Employees\}$$
$$E_8 \stackrel{\text{def}}{=} \{\langle\langle\rangle, \text{"}tup\text{"}, z\rangle \mid z \in Departments\}$$

3. The attributes edges are "parallel" edges. In our example:

$$E_3 \stackrel{\text{def}}{=} \{\langle z, \text{"}Name\text{"}, z\rangle \mid z \in Employees\}$$
$$E_9 \stackrel{\text{def}}{=} \{\langle z, \text{"}DName\text{"}, z\rangle \mid z \in Departments\}$$
$$E_5 \stackrel{\text{def}}{=} \{\langle z, \text{"}Project\text{"}, z\rangle \mid z \in Employees\}$$
$$E_{11} \stackrel{\text{def}}{=} \{\langle z, \text{"}Number\text{"}, z\rangle \mid z \in Departments\}$$

4. Finally, the value edges are also parallel edges, but they contain the real data:

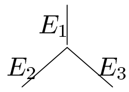
$$E_4 \stackrel{\text{def}}{=} \{\langle z, \pi_1(z), z\rangle \mid z \in \mathit{Employees}\}$$
$$E_{10} \stackrel{\text{def}}{=} \{\langle z, \pi_1(z), z\rangle \mid z \in \mathit{Departments}\}$$
$$E_6 \stackrel{\text{def}}{=} \{\langle z, \pi_2(z), z\rangle \mid z \in \mathit{Employees}\}$$
$$E_{12} \stackrel{\text{def}}{=} \{\langle z, \pi_2(z), z\rangle \mid z \in \mathit{Departments}\}$$

Given a relational database RDB and an UnQL query Q, our translation from UnQL to SQL has three steps. First, we construct the storage schema, Σ, associated with RDB as described above. Second, we translate Q into $\bar{Q}$ on schema Σ, as in Theorem 1, and obtain an output schema Σ'. Third, we reconstruct a relational type from Σ' and derive the SQL query. The last step is more difficult and we describe it in the rest of this section.

Set-edges. Given a storage schema Σ, we say that an edge labeled E is a *single-edge* if the functional dependency $n \to n'$ holds for the triples $(n, l, n') \in E$, otherwise it is a *set-edge*. For instance, the fan-out edges E_2 and E_8 in Figure 4(b) are set-edges, but all others are single-edges. The set-edges in a relational storage schema are always the "tup" edges. But in general, for a derived schema Σ', any edge could be a set edge. Rather than trying to derive a functional dependency from a conjunctive query, we compute the *set-edge* property during schema translation; we omit the details from this abstract. For example, we infer that the edges Q_2 and Q_8 in Figure 5(b) are set edges, while all others are single-edges. We have:

Proposition 2. *Let Σ be a tree storage schema. Then the data graph denoted by Σ is a tuple of flat sets and/or scalar values iff on each path from the root to some leaf in Σ there exists at most one set-edge. Otherwise, Σ denotes data graphs that are nested sets.*

Type reconstruction. After constructing Σ', we must construct valid record types for the output relations. We know that there is at most one set-edge on any path in Σ' from the root to the leaves. We divide all edges into *upper edges*, *set edges*, and *lower edges*: the latter are under some set edge. For example Q_1, Q_7 are the only upper edges in Figure 5(b). The reconstructed type is a tuple of sets and/or scalar values: paths through the upper edges correspond to positions in this tuple. Each set edge denotes a set, whose type we reconstruct as follows. Each lower edge may be labeled by an attribute (like "*Name*" or "*Project*") or by a base type (like *String* or *Int*). In the storage schemas corresponding to relational databases, we always find one attribute edge followed by exactly one value edge. But in general this is no longer the case, because attribute edges and value edges may be intermixed arbitrarily, forming any subtree. We generate a record type with as many fields as there are value edges and generate a field's attribute name by concatenating the labels along the path from the root of the subtree up to the edge. In addition, we generate a field of type *Unit* for each path from the root to a leaf that only contains edges that correspond to labels. For example, consider the following fragment of Σ':



where E_1, E_2, E_3 are sets of triples, E_1 corresponds to an attribute name A, E_2 corresponds to the base type *Int*, and E_3 to the attribute name B. The type reconstruction phase produces the record type $\langle A : \mathit{Int}, A\_B : \mathit{Unit}\rangle$.

Generating an RA query. To generate an RA query, we join the expressions that label the edges of storage schema. For Σ' above, the result expression is:

$$\{\langle A : x, A\_B : \langle\rangle\rangle \mid \langle n_1, A, n_1'\rangle \in E_1, \langle n_2, x, n_2'\rangle \in E_2, \\ \langle n_3, B, n_3'\rangle \in E_3, n_1' = n_2, n_1' = n_3\}$$

Conditionals. Conditionals introduce additional complexity. For every conditional, our translation of schemas produces two subtrees, corresponding to the two branches of the conditional, joined together by two epsilon edges. The first epsilon edge is labeled with a CONJ expression that produces a singleton set when the condition evaluates to true and the empty set when the condition evaluates to false, and vice-versa for the second epsilon edge. Suppose the condition evaluates to true. Then one of the two subtrees will contribute an empty set to the final result. If we apply the above join algorithm, we always get an empty set, which is incorrect. Instead, we choose either one of the sets produced by the two subtrees, depending on the result of the conditional by marking explicitly the edges that come from a conditional. Finally, subtrees in which all the edges are labeled epsilon are eliminated, because they don't correspond to real data.

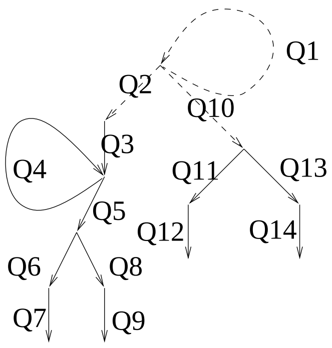
8 Evaluating UnQL Queries in an Integrated System

The main application of our translation technique is the translation of recursive queries on semi-structured data into non-recursive queries on structured, external sources. We illustrate with the example at the end of Section 2. There we integrated a HTML graph with the *Employees* relation, and the corresponding storage schema Σ_2 is in Figure 6 (c). For $E3, E4, E5, E6$, we have similar definitions as for the storage schema in a relational database, see Section 7. $E1$ is given a priori and represents the HTML graph. The interesting part is $E2$, which is populated by edges that link the HTML graph with the relational database. Namely:

$$E2 \stackrel{\text{def}}{=} \{\langle n, \text{"}tup\text{"}, z\rangle \mid \langle n, l, n'\rangle \in E1, z \in \mathit{Employees}, \pi_1(z) = l\}$$

Here l is the label on some hyper-link in the HTML graph, while $\pi_1(z)$ denotes the Name of $z \in \mathit{Employees}$. That is $E2$ will introduce a "tup" edge in the data graph from every HTML page n reachable by a hyper-link l whose label matches the name of some employee z. Note how the storage schema Σ_2 helps us keeping track of the source of the integrated data: the edges populating $E1$ are all HTML hyper-links, those populating $E3, \ldots, E6$ are all from the external source *Employees*, while the edges in $E2$ are making the connection, in effect performing the integration.

After applying query $Q1$, we obtain the storage schema in Figure 13(a). Some of its eighteen relations are defined as the conjunctive queries in Figure 13(b). The important fact to note here is that the external source, *Employees*, is still queried only with conjunctive queries. However, in order to eliminate the ε-edges from the resulting data we need to compute the transitive closure on the $Q1$ relation: this translates into a recursive query on the HTML graph ($E1$).



(a)

$$
\begin{array}{rcl}
Q1(n, \varepsilon, n') & :- & E1(n, \_, n') \\
Q2(n, \varepsilon, n) & :- & E1(n, \_, \_) \\
Q3(n, x, n') & :- & E1(n, x, n'), E1(n', y, n''), \\
 & & \mathsf{matches}\ (y, \text{"}Security\text{"}) \\
Q4, \ldots, Q9 & = & E1, \ldots, E6 \quad \text{identical relations} \\
Q10(n, \varepsilon, n') & :- & E2(n, \_, n') \\
Q11(n, x, n') & :- & E3(n, x, n'), E4(n', y, n''), \\
 & & \mathsf{matches}\ (y, \text{"}Security\text{"}) \\
Q12(n, y, n') & :- & E4(n, y, n'), \mathsf{matches}\ (y, \text{"}Security\text{"}) \\
 & \ldots &
\end{array}
$$

(b)

Fig. 13. (a) Storage schema $\Sigma_2' = \bar{Q}1(\Sigma_2)$. (b) Definitions of Σ_2''s conjunctive queries.

9 Conclusions

We have described a technique for translating queries on semi-structured data into a collection of conjunctive queries. For that we have introduced the notion of storage schema, which is a technique allowing a semi-structured database to be stored as a collection of structured relations. Our main contribution consists in showing how storage schema can be used to translated queries on the semi-structured data into conjunctive queries on these structured relations. Some comments are in order:

1. We are currently working on an implementation of the translation in ML, in conjunction with an UnQL interpreter. We observed that our translation described in Appendix A sometimes introduces an unnecessary level of joins in the resulting conjunctive queries, when translating certain *gext* expressions. As a consequence join-free queries like $Q1$ of Section 2 end up being translated into three-way joins in CONJ. In future work we plan to optimize the translation to avoid generating unnecessary joins.
2. Languages designed for querying semi-structured data are useful also in querying structured data, because they allow us to write concisely browsing queries. We have extended our technique to translate browsing queries into collections of SQL queries. When given as input an UnQL query which is some SQL query Q, we expect the result of the translation to be Q itself (or a syntactically equivalent query). For this we need to implement the optimization described above, else the translated query will have additional, unnecessary joins.
3. Semi-structured data is particularly useful in integrating information sources with various data formats [PGMW95]. The main challenge in such a system is to develop techniques for translating queries on the integrated semi-structured data into queries on the original sources [PGGMU95,PAGM96]. In applications like web site management [FFK+97] one source — the graph of HTML pages — is still unstructured, which requires us to deal with recursive queries. We have briefly described here how storage schemas can address this problem.

References

[Abi97] Serge Abiteboul. Querying semi-structured data. In *ICDT*, 1997.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[AV97] Serge Abiteboul and Victor Vianu. Queries and computation on the web. In *ICDT*, pages 262–275, Deplhi, Greece, 1997. Springer Verlag.

[BDFS97] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *ICDT*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[BDHS96a] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996.

[BDHS96b] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. Technical Report 96-09, University of Pennsylvania, Computer and Information Science Department, February 1996.

[BLS+94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[FFK+97] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL - a web-site management system. In *SIGMOD*, Tucson, Arizona, May 1997.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of VLDB*, September 1996.

$$
\begin{aligned}
E &::= DB \mid t \mid \{\} \mid \{L \Rightarrow E\} \mid E_1 \cup E_2 \mid E_1 \mathbin{@_{\mathcal{X}}} E_2 \mid \mathsf{if}\ B\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 \mid \\
&\qquad X \mid (X_1 := E_1, \ldots, X_m := E_m) \mid \\
&\qquad gext_{\{Z_1 \ldots Z_k\}}(\lambda(l,t).(Z_1 \stackrel{\mathrm{def}}{=} E_1, \ldots, Z_k \stackrel{\mathrm{def}}{=} E_k))(E) \\
B &::= L_1 = L_2 \mid L_1 \neq L_2 \\
L &::= l \mid a
\end{aligned}
$$

Fig. 14. UnCAL's grammar.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Int'l Conf. on Deductive and Object-Oriented Databases*, 1995.

[PGMU95] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *IEEE International Conference on Data Engineering*, pages 132–141, March 1995.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, March 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructure heterogeneous information. In *International Conference on Deductive and Object Oriented Databases*, 1995.

[Won94] Limsoon Wong. *Querying Nested Collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

A Translation of UnCAL into Conjunctive Queries

Review of UnCAL Before describing UnCAL we extend slightly our data model. First we consider an infinite set of *markers* to be given: we denote markers with $X, X_1, Y, Z, \ldots$ For a set of markers $\mathcal{Y} = \{Y_1, \ldots, Y_n\}$ we call a labeled graph *with outputs* $\mathcal{Y}$ a rooted, edge-labeled graph as before, in which some leaf nodes may be labeled with markers from $\mathcal{Y}$. For a set of markers $\mathcal{X} = \{X_1, \ldots, X_m\}$, we call a labeled graph *with inputs* $\mathcal{X}$ an edge-labeled graph as before, which has m distinguished input nodes, associated to the m markers in $\mathcal{X}$, instead of a single root. Combining the two concepts we get labeled graphs with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. The rooted graphs considered before correspond to $\mathcal{X} = \{X_1\}$ and $\mathcal{Y} = \emptyset$.

The grammar in Figure 14 describes the positive fragment of UnCAL: it is the same language as in [BDHS96a], without the non-monotone operator *isempty*. DB denotes the input database, t is a tree variable, $\{\}$ produces the empty graph, $\{L \Rightarrow E\}$ produces a singleton graph, and $E_1 \cup E_2$ produces the union of two rooted graphs. The append operator, $E_1 \mathbin{@_{\mathcal{X}}} E_2$, is the "vertical" equivalent of union: it is assumed that both E_1's outputs and E_2's inputs are $\mathcal{X} = \{X_1, \ldots, X_m\}$.

$$
\begin{array}{lll}
f(\Sigma) & \stackrel{\text{def}}{=} & (Z_1 := f_1(\Sigma), \ldots, Z_k := f_k(\Sigma)) \\
f_i(\{\}) & \stackrel{\text{def}}{=} & \{\} \\
f_i(\{E \Rightarrow \Sigma\}) & \stackrel{\text{def}}{=} & \{\boxed{\{\langle n, \varepsilon, \langle n, l, n' \rangle\rangle \mid \langle n, l, n' \rangle \in E\}} \Rightarrow Z\} \mathbin{@}_Z \qquad // \text{ ``In'' part} \\
& & (Z := tr(\ e_i, z_{\text{new}}, (gen, z_{\text{new}} \in E), \\
& & \qquad (env, [\ l := \boxed{\pi_2(z_{\text{new}})}, \\
& & \qquad\qquad t := replicate(\Sigma, z_{\text{new}}, \boxed{z_{\text{new}} \in E})]))) \mathbin{@}_{\mathcal{Z}} \\
& & (Z_1 := \{\boxed{\{\langle \langle n, l, n' \rangle, \varepsilon, n' \rangle \mid \langle n, l, n' \rangle \in E\}} \Rightarrow Z_1\},\ // \text{ ``Out''} \\
& & \ldots \qquad\qquad\qquad\qquad\qquad\qquad // \text{ part} \\
& & Z_k := \{\boxed{\{\langle \langle n, l, n' \rangle, \varepsilon, n' \rangle \mid \langle n, l, n' \rangle \in E\}} \Rightarrow Z_k\}), \\
f_i(\Sigma \cup \Sigma') & \stackrel{\text{def}}{=} & f_i(\Sigma) \cup f_i(\Sigma') \\
rep(E, z, gen) & \stackrel{\text{def}}{=} & \boxed{\{\langle \langle z, n \rangle, l, \langle z, n' \rangle\rangle \mid gen, \langle n, l, n' \rangle \in E\}} \\
replicate(\Sigma, z, gen) & \stackrel{\text{def}}{=} & map(\lambda(E).rep(E, z, gen))(\Sigma)
\end{array}
$$

Fig. 15. Translation of *gext*.

Then the result is the graph E_1 in which each leaf marked X_i in E_1 is "replaced" by the input node X_i of E_2. if is a conditional: it can only compare for equality or inequality two label variables and/or constants. X is the same as the empty graph but with the unique node labeled as output X. $(X_1 := E_1, \ldots X_m := E_m)$ constructs a graph with inputs $\{X_1, \ldots, X_m\}$. Finally *gext* allows us to define k mutually recursive functions as in Figure 8. Namely $gext_{\mathcal{Z}}(\lambda(l,t).(Z_1 := E_1, \ldots, Z_k := E_k))(E)$ defines k recursive functions $g_1, \ldots, g_k$ as in Figure 8, in which each expression F_i is:

$$F_i(l, t, g_1(t), \ldots, g_k(t)) \stackrel{\text{def}}{=} E_i(l,t) \mathbin{@}_{\mathcal{Z}} (Z_1 := g_1(t), \ldots, Z_k := g_k(t))$$

By imposing such restrictions on form of the recursive functions we can guarantee well definedness on cyclic graphs, see [BDHS96b]. For the purpose of this paper we define the meaning of the *gext* expression to be $g_1(E)$. For example, query $Q1$ of Section 2 is equivalent to the following *gext* expression:

$$
\begin{array}{l}
gext_{X_1}(\lambda(x,t).(X_1 := X_1 \cup gext_{X_2}(\lambda(y,t').(X_2 := \textsf{if } (y \textsf{ matches } \text{"}Security\text{"}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsf{then } \{x \Rightarrow \{y \Rightarrow t'\}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsf{else } \{\}))(t)))(DB)
\end{array}
$$

Translating queries with free variables Let us review how we compute $\Sigma' = gext(\lambda(l,t).(Z := e_1))(\Sigma)$. As illustrated in Section 6, we obtain Σ' by translating e_1 once for every edge labeled $E2$ in Σ. But the translation of e_1 cannot be a

UnCAL expression e	$\Sigma = tr(e, z, gen, env)$	Comments
DB	$replicate(DB, z, gen)$	*replicate* defined in Fig. 15
t	$env(t)$	
$\{\}$	$\{\}$	
$\{l \Rightarrow e_1\}$	$\{rep(\boxed{\{\langle\langle\rangle, env(l), \langle\rangle\rangle\}}, z, gen) \Rightarrow \Sigma_1\}$	*rep* defined in Fig. 15
$e_1 \cup e_2$	$\{rep(\boxed{\{\langle\langle\rangle, \varepsilon, \langle\rangle\rangle\}}, z, gen) \Rightarrow \Sigma_1,$ $rep(\boxed{\{\langle\langle\rangle, \varepsilon, \langle\rangle\rangle\}}, z, gen) \Rightarrow \Sigma_2\}$	
$e_1 \ @_{y_1}\ e_2$	$\Sigma_1 \ @_{y_1}\ \Sigma_2$	
if $l = l'$ then e_1 else e_2	$\{\ rep(\boxed{\{\langle\langle\rangle, \varepsilon, \langle\rangle\rangle\}}, z,$ $\boxed{gen, (env(l) = env(l'))}) \Rightarrow \Sigma_1,$ $rep(\boxed{\{\langle\langle\rangle, \varepsilon, \langle\rangle\rangle\}}, z,$ $\boxed{gen, (env(l) \neq env(l'))}) \Rightarrow \Sigma_2\}$	
Y	Y	
$(X_1 := e_1, \ldots, X_m := e_m)$	$(X_1 := \Sigma_1, \ldots X_m := \Sigma_m)$	
$gext_{\mathcal{Z}}(\lambda(l, t).(\ Z_1 := e_1,$ $\ldots,$ $Z_k := e_k))(e')$	$f(\Sigma')$	$\Sigma' = tr(e', z, gen, env)$; f defined in Figure 15

Fig. 16. Translation of UnCAL.

normal storage schema, because e_1 may refer to free variables, such as l and t. We handle free variables as follows. Consider a label variable l (it can be introduced only in a *gext* construct). Suppose l appears in e_1 as $\{l \Rightarrow \ldots\}$. Taken in isolation, this denotes a single edge, which we would construct as $E1 = \{\langle\langle\rangle, l, \langle\rangle\rangle\}$, but taken in context, that edge has to be replicated once for every element in the set denoted by $E2$. Hence, we actually generate the set $\{\langle z, \pi_2(z), z\rangle \mid z \in E2\}$. For an analogy, consider a subexpression $\{a \Rightarrow \ldots\}$ in e_1, where a is a label constant. In isolation, this should generate the CONJ expression $E1 = \{\langle\langle\rangle, a, \langle\rangle\rangle\}$, but in context, we have to replicate this edge once for every element in $E2$, so the correct expression is $\{\langle z, a, z\rangle \mid z \in E2\}$. We cannot collapse these identical edges into one, because they may lead to distinct subgraphs underneath. To translate an expression e_1 that refers to free variables, we need the additional information:

- The CONJ variable name, z.
- The scalar CONJ expression which replaces l, in our case $\pi_2(z)$. This requires an *environment*, which maps label variables to CONJ scalar expressions

and maps tree variables to storage schemas. It is denoted by $env = [l_1 := E_1, \ldots, l_n := E_n, t_1 := \Sigma_1, \ldots, t_m := \Sigma_m]$, and for a label variable l_i, $env(l_i)$ denotes its associated value, E_i, while for a tree variable t_j, $env(t_j)$ denotes Σ_j. When a is a label constant, we define $env(a) = a$.
– The *generator* $z \in E2$. In general, a generator is a sequence $C_1, \ldots, C_n$ where each C_i is either of the form $z \in E$, or a condition of the form $E = E'$ or $E \neq E'$ (with E, E' scalar expressions). The latter are introduced by the translation of if expressions.

The translation The translation function is described in Figures 16 and 15. Here $tr(e, z, gen, env)$ takes a UnCAL expression e, a CONJ variable z, a generator gen, and an environment env, and returns the storage schema Σ. The function is recursive on the structure of e: whenever e has subexpressions $e_1, e_2, \ldots$, we assume that $\Sigma_i = tr(e_i, z, gen, env)$, for $i = 1, 2, \ldots$ The translation begins with $tr(e, z, \boxed{z \in \{\langle\rangle\}}, [])$, where $[]$ is the empty environment. The auxiliary function $rep(E, z, gen)$ takes a CONJ expression E denoting a set of edges $\langle n, l, n' \rangle \in E$ and "replicates" them according to the generator gen (see Fig. 15). E may actually have free variables bound by the generator, hence it is placed after gen in Fig. 15. The function $replicate(\Sigma, z, gen)$ does this for an entire storage schema Σ, by replacing every label E with $rep(E, z, gen)$: here $map(f)(\Sigma)$ returns a storage schema Σ' isomorphic to Σ, in which every label E is replaced with $f(E)$.

The only non-trivial part of tr is in the translation of a $gext(\ldots)(e')$ expression. As explained earlier, we translate e' to get Σ, then we translate the body of $gext$ several times, once for every CONJ expression E occurring in Σ'. This is done by the function f, described in Figure 15, which actually consists of a number of recursive functions f_i, $i = 1, k$, where k is the number of markers in $gext$. The role of the f_i is to traverse the storage schema Σ', process each expression E in Σ', and replace E with a storage schema consisting of three parts: the "in" part, a translation of the body of $gext$, and the "out" part. It is in the translation of the body where the generator and environment are expanded; z_{new} is a fresh CONJ variable.

We extend the convention that the root of the data graph is the empty tuple, $\langle\rangle$, to the following invariant: the translation $tr(\Sigma, z, gen, env)$ denotes a family of data graphs, whose roots are $\{\langle z, \langle\rangle\rangle \mid gen\}$. Applied to the top translation, we obtain as root $\langle\langle\rangle, \langle\rangle\rangle$, instead of the desired $\langle\rangle$: a minor post-processing is necessary. We summarize here, and defer the proof to a full version of the paper:

Theorem 3. *Let Σ be a storage schema and e be some UnCAL expression. Then (1) $\Sigma' = tr(e, z, \boxed{z \in \{\langle\rangle\}}, [])$ is type correct, and (2) whenever DB is a data graph denoted by the storage schema Σ, and DB' is the result of computing e, then the data-graph denoted by Σ' is bisimilar to e.*