A Query Language for a Web-Site Management System

Mary Fernandez* Daniela Florescu* Alon Levy* Dan Suciu*

1 Introduction

We have designed a system, called STRUDEL, which applies familiar concepts from database management systems, to the process of building web sites. The main motivation for developing STRUDEL is the observation that with current technology, creating and managing large sites is tedious, because a site designer must simultaneously perform (at least) three tasks: (1) choosing what information will be available at the site, (2) organizing that information in individual pages or in graphs of linked pages, and (3) specifying the visual presentation of pages in HTML. Furthermore, since there is no separation between the physical organization of the information underlying a web site and the logical view we have on it, *changing* or *restructuring* a site are unwieldy tasks.

In STRUDEL, the web site manager can separate the logical view of information available at a web site, the structure of that information in linked pages, and the graphical presentation of pages in HTML. First, the site builder defines independently the data that will be available at the site. This process may require creating an *integrated* view of data from multiple (external) sources. Second, the site builder defines the *structure* of the web-site. The structure is defined as a *view* over the underlying information, and different versions of the site can be defined by specifying multiple views. Finally, the graphical representation of the pages in the web site is specified.

This paper describes the query language that lies at the heart of the STRUDEL system. In STRUDEL, we model the data at the different levels as graphs. That is, the data in the external sources, the data in the integrated view and the web-site itself are modeled as graphs. A graph model is appropriate because site data may be derived from multiple sources, such as existing database systems and HTML files. Consequently, our system requires a query language for (1) defining the integrated view of the data, and (2) defining the structure of web sites. An important requirement of our query language is that it be able to *construct* graphs. Our query processor needs to be able to answer queries that involve accessing different data sources. Even though we model the sources as containing graphs, we cannot assume they have a *uniform* representation of graphs. Hence, our query processor needs to *adhere* to possible limitations on access to data in the graphs, and should be able to *exploit* additional querying capabilities that an external source may have. We have designed a general framework for processing STRUDEL queries over multiple unstructured data sources, and are designing optimizations that use the capabilities of external sources whenever possible.

The purpose of this paper is to describe the syntax and semantics of STRUQL, the query language at the core of STRUDEL. We believe that STRUQL is a language of independent interest, and is useful for other applications involving the management of semistructured data, as well as a view definition language for such data. We discuss the relationship of STRUQL to other languages proposed in the literature in Section 6: see [Abi97, Bun97].

2 STRUDEL Architecture

In every level of the STRUDEL system, data is viewed uniformly as a graph. At the bottom-most level, data is stored in STRUDEL's *data graph* repository or in external sources. External sources may have a variety of formats, but each is translated into the graph data model by a wrapper (see Figure 1). STRUDEL's graph model is similar to that of OEM [PGMW95]. A data graph contains *objects* connected by directed edges

*AT&T Labs — Research, email: {mff,dana,levy,suciu}@research.att.com

labeled with string-valued *attribute names*. Objects are either nodes, carrying a unique object identifier (oid), or are atomic values, such as integers, strings, files, etc. STRUDEL also provides named *collections* of objects, i.e., sets of oids: technically speaking these are redundant, since every collection can be represented as a wide subtree, but we included them in the data model for convenience.

The *data graph* describes the logical structure of all the information available at that site, and may be obtained by integrating information from the various external sources. This integration is done in a similar way to recently proposed data integration prototypes such as Tsimmis [PGMW95] and the Information Manifold [LRO96]. Given the data graph, a site builder can define one or more *site graphs*; each site graph represents the logical structure of the information displayed at that site (i.e., a node for every web page and attributes describing the information in the page and links between pages). There can be several site graphs, corresponding to different versions (or views) of the web site. Finally, the HTML generator constructs a browsable HTML graph from a graph site. The HTML generator creates a web page for every node in the site graph, using the values of the attributes of the node.

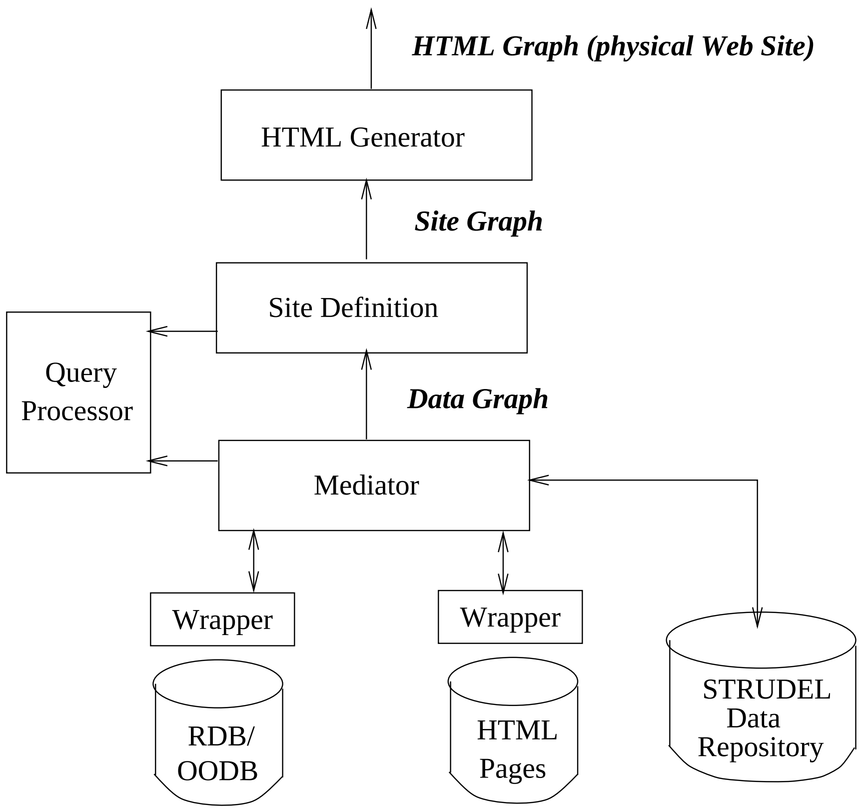


Figure 1: STRUDEL Architecture

In STRUDEL, we need to query and/or to transform graphs: (1) at the *integration* level, when data from different external sources is integrated into the data graph, and (2) at the *site-graph definition* level, when site graphs are constructed from a data graph. In addition, STRUDEL's design enables us to provide an interface for ad-hoc queries over a web site. We use the same query and transformation language, STRUQL(Site TRansformation Und Query Language), at all three levels. We describe STRUQL's core fragment next.

3 Data Model and Query Language

Data Model. Our data model is a minor variation on the graph model OEM [PGMW95], designed for semistructured data. First we fix a universe of *values*, and one of *labels*. All integers, reals, strings, *true* and $false$ are values, while labels are typically strings. A *database* is a directed, finite graph, whose set of nodes consists of a finite set of oids and a finite set of values. Edges can go from oids to oids, and from oids to values, but an edge cannot emanate at a value. Labels are attached to edges: for any two nodes x, y and any label a there can be at most one edge between x and y labeled a; when that's the case we write $x \rightarrow a \rightarrow y$. In addition to the graph, the database also contains a number of *collections*. Each collection is a set of nodes. Intuitively, the collections are the entry points to the graph. For example, the graph may have a collection $Root(x)$, consisting of a single node x which is the graph's root. There is an implicit schema associated to that database, namely the number and names of the collections.

Query Language: Basic Syntax. To start, consider the following STRUQL query which returns all PostScript papers directly accessible from home pages:

where $HomePages(p), p \rightarrow "Paper" \rightarrow q, isPostScript(q)$
collect $PostscriptPages(q)$

Here $HomePages$ is a collection, $"Paper"$ is an edge label, and $isPostScript$ is a predicate testing whether node q is a PostScript file. The condition $p \rightarrow "Paper" \rightarrow q$ means that there exists an edge labeled $"Paper"$ from p to q. The query constructs a new collection, $PostscriptPages$, consisting of all answers.

STRUQL is novel, in the way it combines regular expressions with the creation of new graphs from existing graphs; its create, and link clauses specify new graphs. The following example copies the input graph and adds a $"Home"$ edge from each node back to the root:

where $Root(p),\ p \rightarrow * \rightarrow q,\ q \rightarrow l \rightarrow q'$
create $N(p),\ N(q),\ N(q')$
link $N(q) \rightarrow l \rightarrow N(q'), N(q) \rightarrow "Home" \rightarrow N(p)$
collect $NewRoot(N(p))$

The $*$ in $p \rightarrow * \rightarrow q$ denotes a regular path expressions: in this simple case it means any path from p to q. N is a Skolem function creating new oids. The query first finds all nodes q reachable from the root p (including p itself) and all nodes q' directly accessible from q by one link labeled l. Then it constructs new nodes $N(q)$ and $N(q')$: in effect, this copies all nodes accessible from the root. The query adds a link l between any pair of nodes that were linked in the original graph and adds a new $Home$ link that points to the new root. Finally, it creates an output collection $NewRoot$ that contains the new graph's root.

A similar query produces a site graph, i.e., a view of the input graph, called *TextOnly*, that excludes any nodes that contain image files:[1]

where $Root(p),\ p \rightarrow * \rightarrow q,\ q \rightarrow l \rightarrow q'$, not $(isImageFile(q'))$
create $N(p),\ N(q),\ N(q')$
link $N(q) \rightarrow l \rightarrow N(q')$,
collect $TextOnlyRoot(N(p))$

In the general syntax, STRUQL has four clauses, select, create, link, collect whose syntax is given below:

where $C_1, \ldots, C_k$
create $N_1, \ldots, N_n$
link $L_1, \ldots, L_p$
collect $G_1, \ldots, G_q$

The C's in the where clause are called *conditions* and are given by the grammar:

$$\begin{array}{rcl} C & ::= & PathCond \mid BoolCond \\ PathCond & ::= & NodeExpr \rightarrow RPE \rightarrow NodeExpr \end{array}$$

[1]This example is inspired by an inconsistency in the CNN web site `http://www.cnn.com`. The site provides a link to a text-only version. But, surprisingly, by following the links from that page one ends up again at pages with images.

That is, a condition can be either a *path condition*, or a *boolean condition*. Path conditions are regular path expressions, RPE between two nodes:

$$RPE ::= \mathit{LabelConst} \mid \mathit{LabelVar} \mid \mathit{UnaryBoolCond} \mid \text{“\_“} \mid (RPE.RPE) \mid (RPE\text{“}|\text{“}RPE) \mid RPE\text{“}*\text{“} \mid RPE\text{“}+\text{“}$$

We use quotation marks here to distinguish the syntax from the meta syntax. Here *UnaryBoolCond* is a boolean combination of user-defined external functions on labels. For example the path condition $x \rightarrow (\mathit{isMyEdge})+ \rightarrow y$ uses the user-defined function $\mathit{isMyEdge}(a)$, and is satisfied whenever there exists a path from x to y whose labels form the sequence $a_1, \ldots, a_n$ and the following hold: $n \geq 1$, $\mathit{isMyEdge}(a_1)$, ..., $\mathit{isMyEdge}(a_n)$. The wild-char _ denotes any label (and is the same as the predicate *true*). We abbreviate $(\_)*$ with $*$ and $(\_)+$ with $+$, thus writing $x \rightarrow * \rightarrow y$ instead of $x \rightarrow (\_)* \rightarrow y$.

A *BoolCond* is an arbitrary boolean condition on nodes, values, and labels. Atomic boolean conditions are collection memberships, like $\mathit{Root}(x)$, built-in predicates, like $x < y$, or user defined predicates, like $\mathit{isPostScript}(x)$. Note that we cannot negate path conditions, i.e. the query where $\mathit{Root}(x)$, not $(x \rightarrow * \rightarrow y)$ is not legal. As explained below we *can* express this query in StruQL, using composed queries.

Node expressions occur in all three clauses create, link, and collect. Node expressions are either (1) node variables, or (2) Skolem terms. The latter are obtained by applying a Skolem function (of any arity) to either of the following: node expressions, label variables, or values. In the create clause each of $N_1, \ldots, N_n$ is a Skolem term. In the link clause, each L is of the form $NE1 \rightarrow VC \rightarrow NE2$, where VC is a label variable or constant, $NE1$ is a Skolem term, and $NE2$ a node expression. Finally, each G in the collect clause is of the form *CollectionName*(*NodeExpr*).

Query Blocks StruQL queries are typically larger than OLTP or Decision Support database queries, because they have to construct new graphs, with a diversity rich enough to please a human viewer. For that reason we allow the where, create, link, and collect clauses to be interleaved, and introduce some block structure into the language. We give the syntax below. There is one[2] named *input graph* and one named *output graph* per query:

```
Block ::= ( where   C1, ..., Ck,
            [create N1, ..., Nn]
            [link   L1, ..., Lp]
            [collect G1, ..., Gq]
            ["{"Block, ..., Block"}"])*
Query ::= input ident
          Block
          output ident
```

Here $(\ldots)*$ means repetition and $[\ldots]$ means that ... is optional. For example the following query copies that portion of the graph named *DataGraph* which is reachable only through attributes *“Paper”*, *“TechReport”*, *“Title”*, *“Abstract”*, *“Author”*. In addition it constructs a special node *Authors*() and connects it to all pages corresponding to *“Author”*s. The output graph is called *SiteGraph*. One way to write this in StruQL is:

```
input   DataGraph
where   Root(x), x → * → y, y → l → z,
        l in { "Paper", "TechReport", "Title", "Abstract", "Author"}
create  Authors(), Page(y), Page(z)
link    Page(y) → l → Page(z)
where   x → * → y1, y1 → "Author" → z1
link    Authors() → "Author" → Page(z1)
output  SiteGraph
```

[2]In order to integrate information from several source, we allow multiple input graphs. When multiple input graphs are present, every occurrence of a collection needs to be preceded by a graph name. For clarity of presentation however, we focus on queries with only one input graph.

Intermixing the where, create, link clauses makes the query easier to read. This is nothing more than syntactic convenience, since the meaning is the same as that of the query in which all clauses are joined together:

input $\mathit{DataGraph}$
where $\mathit{Root}(x), x \rightarrow * \rightarrow y, y \rightarrow l \rightarrow z,$
l in $\{$"Paper", "TechReport", "Title", "Abstract", "Author"$\}$
$x \rightarrow * \rightarrow y1, y1 \rightarrow$ "Author" $\rightarrow z1$
create $\mathit{Authors}(), \mathit{Page}(y), \mathit{Page}(z)$
link $\mathit{Page}(y) \rightarrow l \rightarrow \mathit{Page}(z)$
$\mathit{Authors}() \rightarrow$ "Author" $\rightarrow \mathit{Page}(z1)$
output $\mathit{SiteGraph}$

There is a certain sense that the variables $y1, z1$ are redundant: indeed any valid binding of $y1, z1$ is also a valid binding of y, z. But the converse is not true, and in fact there is a special action done for the variables $y1, z1$ which is not done for any y, z. This makes the query harder to read. By using blocks we can avoid having to introduce the new variables:

input $\mathit{DataGraph}$
where $\mathit{Root}(x), x \rightarrow * \rightarrow y, y \rightarrow l \rightarrow z,$
l in $\{$"Paper", "TechReport", "Title", "Abstract", "Author"$\}$
create $\mathit{Authors}(), \mathit{Page}(y), \mathit{Page}(z)$
link $\mathit{Page}(y) \rightarrow l \rightarrow \mathit{Page}(z)$
$\{$ where $l =$ "Author"
link $\mathit{Authors}() \rightarrow$ "Author" $\rightarrow \mathit{Page}(z)$
$\}$
output $\mathit{SiteGraph}$

The semantics of queries with nested blocks can be reduced to that of queries without nested blocks. For example each query of the form:

where $E(x, y, z)$
create $C(x, y, z)$
link $L(x, y, z)$
$\{$ where $E1(x, y, z, u, v)$
create $C1(x, y, z, u, v)$
link $L1(x, y, z, u, v)$
$\}$

where $E, C, L, E1, C1, L1$ are expressions for the corresponding clause, is equivalent to the following query without block structure.

where $E(x, y, z), E(x1, y1, z1), E1(x1, y1, z1, u, v)$
create $C(x, y, z), C1(x1, y1, z1, u, v)$
link $L(x, y, z), L(x1, y1, z1, u, v)$

Finally we note that the query with block structure is easier to evaluate than that without. A query processor would need to discover the variable redundancy in the query without block structure (that every binding of $x1, y1, z1$ is a binding for x, y, z too).

Query Composition We can express query composition in StruQL by replacing the graph name in input with another StruQL query. Recall that where $\mathit{Root}(x), x \rightarrow * \rightarrow y$, not $(x \rightarrow$ "A"$* \rightarrow y)$ collect $C(y)$

is incorrect, because we do not allow negations on path conditions. However we can express that query as a composition of two StruQL queries. For example, if the collection *Root* is guaranteed to contain exactly one element, then the following is a correct translation:

```
input ( input G
        where Root(x), x → "A"* → y
        collect D(y))
where Root(x), x → * → y, not (D(y))
collect C(y)
```

The general case requires a more involved translation.

4 StruQL Semantics

StruQL's semantics can be described in two stages. The *query stage* depends only on the where clause and produces all possible bindings of variables to values that satisfy all conditions in the clause; its result is a relation with one column for each node or label variable in the where clause. The *construction* stage constructs a new graph from this relation, based on the create, link, collect clauses. We explain the details next.

We adopt active-domain semantics for StruQL. For a data graph G, let O be the set of all oids and values in the graph, and L be the set of all labels in G. Let V be the set of all node and label variables in a query. The meaning of the where-clause is the set of assignments $\theta : V \to O \cup L$ that satisfy all conditions in the where clause. Each assignment maps node variables to O and edge variables to L. The meaning of the create −link −collect clauses is as follows. First, the create clause specifies what new nodes to create: for each row in the relation, one new node is created, corresponding to each Skolem term in the create clause. Second, the link clause specifies what edges to construct in the output graph: that is, an edge is created for every triplet in the link clause. Finally, the collect clause places nodes in the newly defined collections.

Two comments are in order. First, notice that, when a Skolem function is applied to the same arguments stemming from two different rows, then the same node is returned. Second, conceptually, the result of the query is a new graph, consisting of: (1) a fresh copy of the old graph, and (2) all the new nodes, links, and collections created explicitly in the query. Thus, edges in the link clause pointing "back" to the old graph are actually pointing to the fresh copy. Furthermore, the collections of the new graph are precisely those defined in collect.

However, the active-domain semantics is unsatisfactory because it depends on how we define the active domain; the semantics changes if, for example, we compute the active domain only for the *accessible* part of the graph. The situation is similar to the domain independence issue in the relational calculus: there it is solved by considering *range-restricted* queries, which are guaranteed to be domain independent, i.e., their semantics does not change if we artificially change the active domain. We are currently specifying range-restriction rules for StruQL.

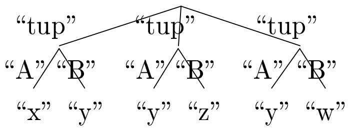
5 Expressive power

StruQL's regular path expressions, like those in LOREL and UnQL, require graph traversal and, therefore, the computation of transitive closure. The ability to compute the transitive closure of an input graph does not imply the ability to compute the transitive closure of an *arbitrary* binary or $2n$-ary relation. This is proven formally for UnQL [BDHS96]. Surprisingly, StruQL *can* express transitive closure of an arbitrary relation as the composition of two queries[3]. For example, consider the tree-encoding of a binary relation $R(A, B)$ with attributes A and B, as shown below. We can compute all nodes reachable from "x" with two StruQL queries. The first constructs the graph corresponding to the relation $R(A, B)$, and the second uses the regular expression $*$ to find all nodes accessible from the root.

[3]It follows from the result in [BDHS96] that a single where −link query cannot express transitive closure.

input (where $Root(r), r \rightarrow "tup" \rightarrow s1, r \rightarrow "tup" \rightarrow s2,$
$s1 \rightarrow "A" \rightarrow x1, s1 \rightarrow "B" \rightarrow y1$
$s2 \rightarrow "A" \rightarrow x2, s2 \rightarrow "B" \rightarrow y2$
$y1 = x2$
create $N(y1), N(x2)$
link $N(y1) \rightarrow "bogus" \rightarrow N(x2)$
collect $NewRoot(N("x")))$
where $NewRoot(x), x \rightarrow * \rightarrow N(y)$
collect $Result(y)$

A	B
"x"	"y"
"y"	"z"
"y"	"w"



We can prove that StruQL has precisely the same expressive power as first order logic extended with transitive closure [Imm87], FO+TC. That is, considering a two-sorted (with sorts Oid and $Label$), first order vocabulary $\tau = \{E, C_1, \ldots, C_k\}$, where $E(Oid, Label, Oid)$ is the edge relation and $C_1(Oid), \ldots, C_k(Oid)$ are the collections, then a boolean query over this vocabulary is expressible in pure $StruQL$ (i.e. without external predicates) if and only if it is expressible in FO + TC.

6 Related Languages

Several languages have been developed for querying and restructuring graph and semistructured data. For example, the LOREL language [QRS+95, AGM+97] has been developed in the Tsimmis project for the application of data integration. In comparison to StruQL, LOREL has the equivalent expressive power to the **where** clause of StruQL, but unlike LOREL, StruQL can construct an arbitrary new output graph (with the **create** and **link** clauses). This feature is strictly necessary in the application of creating web sites.

UnQL [BDHS96], another query language for semistructured data, can construct arbitrary new graphs. However, as explained above, StruQL is more expressive than UnQL: the latter cannot compute transitive closure of an arbitrary $2n$-ary relation.

In theory, StruQL has precisely the same expressive power as stratified linear datalog. However the translation of StruQL queries into stratified linear datalog results in cumbersome and hard to understand queries. In particular, StruQL enables a concise representation of regular path expressions and clearly separates the querying and the creation of a graph creation.

GraphLog [CEH+94] is another query language designed for general purpose database applications, succeeding G and $G+$ [CMW87, CMW88, Woo88]. GraphLog combines datalog notation with a visual query language and has the same expressive power as stratified linear datalog.

References

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

[AGM+97] Serge Abiteboul, Roy Goldman, Jason McHugh, Vasilis Vassalos, and Yue Zhuge. Views for semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996.

[Bun97] Peter Buneman. Tutorial: Semistructured data. In *PODS*, 1997.

[CEH+94] M.P. Consens, F.Ch. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista. Architecture and applications of the hy+ visualization system. *IBM Systems Journal*, 33:3:458–476, 1994.

[CMW87] I. Cruz, A.O. Mendelzon, and P.T Wood. A graphical query language supporting recursion. In *Proceedings of ACM SIGMOD Conf.*, San Francisco, California, May 1987.

[CMW88] I. Cruz, A.O. Mendelzon, and P.T Wood. G+: recursive queries without recursion. In *Proc. Second Int'l Conf. on Expert Database System*, Tysons Corner, Virginia, April 1988.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference, Bombay, India.*, 1996.

[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.

[Woo88] Peter T. Wood. *Queries on Graphs*. PhD thesis, University of Toronto, Toronto, Canada, M5S 1A1, December 1988. Available as University of Toronto Technical Report CSRI-223.