Catching the Boat with Strudel: Experiences with a Web-Site Management System

Mary Fernández
AT&T Labs
mff@research.att.com

Daniela Florescu*
Inria Roquencourt
dana@rodin.inria.fr

Jaewoo Kang
Savera Systems
kang@savera.com

Alon Levy
Univ. of Washington
alon@cs.washington.com

Dan Suciu
AT&T Labs
suciu@research.att.com

Abstract

The Strudel system applies concepts from database management systems to the process of building Web sites. Strudel's key idea is separating the management of the site's data, the creation and management of the site's structure, and the visual presentation of the site's pages. First, the site builder creates a uniform model of all data available at the site. Second, the builder uses this model to declaratively define the Web site's structure by applying a "site-definition query" to the underlying data. The result of evaluating this query is a "site graph", which represents both the site's content and structure. Third, the builder specifies the visual presentation of pages in Strudel's HTML-template language. The data model underlying Strudel is a semi-structured model of labeled directed graphs.

We describe Strudel's key characteristics, report on our experiences using Strudel, and present the technical problems that arose from our experience. We describe our experience constructing several Web sites with Strudel and discuss the impact of potential users' requirements on Strudel's design. We address two main questions: (1) when does a declarative specification of site structure provide significant benefits, and (2) what are the main advantages provided by the semi-structured data model.

1 Introduction

The World-Wide Web (WWW) has become a prime vehicle for disseminating information. As a result, the number of large Web sites with complex structure and that serve information derived from multiple data sources is increasing. Managing the *content* and the *structure* of such Web sites presents a novel data management problem.

To understand the problem, consider a Web-site builder's tasks: (1) choosing and accessing the data that will be displayed at the site, (2) designing the site's structure, i.e., specifying the data contained within each page and the links between pages, and (3) designing the visual presentation of pages. In existing Web-site management tools, these tasks are, for the most part, interdependent. Without any site-creation tools, a site builder writes HTML files by hand or writes programs to produce them and must focus simultaneously on a page's content, its relationship to other pages, and its visual presentation. As a result, several important tasks, such as automatically updating a site, restructuring a site, or enforcing integrity constraints on a site's structure, are tedious to perform. To support these tasks naturally, we view the problem from a data management perspective.

We have developed the STRUDEL system [12], which applies concepts from database management systems to Web-site creation and management. In particular, STRUDEL supports declarative specification of a Web site's content and structure and automatically generates a browsable Web site from a specification. STRUDEL's key idea is separating the management of a Web site's data, the management of the site's structure, and the visual presentation of the site's pages.

Using STRUDEL, the site builder first creates an integrated view of the data that will be available at the site. The Web site's raw data resides either in external sources (e.g., databases, structured files) or in STRUDEL's internal data repository. In STRUDEL's mediator component, as in all of its other components, all external and internal data is modeled as a labeled directed graph, which is the model commonly used for semistructured data [1, 6]. A set of source-specific wrappers translates the external representation into the graph model. The integrated view of the data is called the *data graph*. Second, the site builder declaratively specifies the Web site's structure using a *site-definition query* in STRUQL, STRUDEL's query language. The result of evaluating the site-definition query on the data graph is a *site graph*, which models both the site's content and structure. Third, the builder specifies the visual presentation of pages in STRUDEL's HTML-template language. The HTML generator produces HTML text for every node in the site graph from a corresponding HTML template; the result is the browsable Web site.

STRUDEL is based on a semistructured data model of labeled, directed graphs. This model was introduced to manage *semistructured* data, which is characterized as having few type constraints, irregular structure, and rapidly evolving or missing schema [1, 6]. This data model was appealing for STRUDEL, because Web sites are graphs with irregular structure and non-traditional schemas. Furthermore, semistructured data facilitates integration of data from multiple, non-traditional sources.

[*] Research done while authors Florescu, Kang, and Levy were at AT&T Labs.

Strudel provides several benefits. Since a Web site's structure and content are defined declaratively by a query, not procedurally by a program, it is easy to create multiple *versions* of a site. For example, it is possible to build internal and external views of an organization's site or to build sites tailored to novice or expert users. Currently, creating multiple versions requires writing multiple sets of programs or manually creating different sets of HTML files. In Strudel, a site builder produces multiple sites by applying different site-definition queries to the same underlying data or by creating multiple HTML renderings of the same site graph. Strudel's architecture also supports evolution of a Web site's structure. For example, to reorganize pages based on frequent usage patterns or to extend the site's content, we simply rewrite the site-definition query. Declarative specification of Web sites can offer other advantages. For example, it becomes possible to express and enforce integrity constraints on the site and to update a site incrementally when changes occur in the underlying data.

Strudel clearly separates the three tasks of building Web sites and is the first system that supports declarative specification of a site's content *and* structure. Other recent research prototypes support the separation of the three tasks, but do not support declarative specification of content or structure [5, 22]. Other research projects support declarative specification, but merge the tasks [3, 10]. Commercial tools such as Vignette's StoryServer and those provided by major database vendors separate the management of the underlying data from its visual presentation. Individual pages or sets of related pages are constructed dynamically by evaluating queries that are embedded in HTML templates; query results are merged into HTML templates to produce pages. Other products provide graphical user interfaces that support drag-and-drop editing of individual pages (e.g., Microsoft's FrontPage, NetObjects' Fusion) or of the structure between individual pages (e.g., Elemental's Drumbeat).

This intense activity in research and industry indicates that Web-site management is an important problem, and because its central issue is management of site content and structure, it should be of interest to the database community. Given this, our goal is to gain experience quickly using Strudel so that we may understand which aspects of Web-site management benefit most from application of database concepts and identify the critical research issues we should focus on in this area. In this paper, we describe our experience constructing several Web sites with Strudel and discuss the impact of users' requirements on Strudel's design. Based on this experience, our study answers two main questions:

- Is separating the three tasks of Web-site creation natural in practice and under what circumstances does declarative specification of site structure provide significant benefits?
- Which characteristics of the semistructured data were most important in Strudel and what prevented us from using a traditional data model?

We first describe Strudel's architecture and our design choices and present the data management problems that arise when building complex Web sites. We also describe two technical problems that arose from our experience and that we solved in Strudel. First, we observed that in some cases a site's structure can be encoded either in the site graph or in the visual presentation. This led us to develop Strudel's HTML template language, whose functionality overlaps the StruQL query language. This overlap permits users to encode their sites in whatever way is natural. Second, StruQL permits the site builder to construct fragments of a Web site separately using multiple queries. To view a site's complete, abstract structure, we generate a *site schema* from the site's StruQL queries. A site schema represents a StruQL query as a labeled directed graph and can be viewed as a *schema* of the Web sites that result from evaluating the StruQL query. Site schemas allow the user to view the site's abstract structure during design. More importantly, they are the basis for an algorithm that verifies whether a given set of integrity constraints on a Strudel-generated site is guaranteed to be satisfied [14] and for an incremental-evaluation algorithm that converts one site-definition query into multiple queries that are evaluated dynamically when a user browses the site [15].

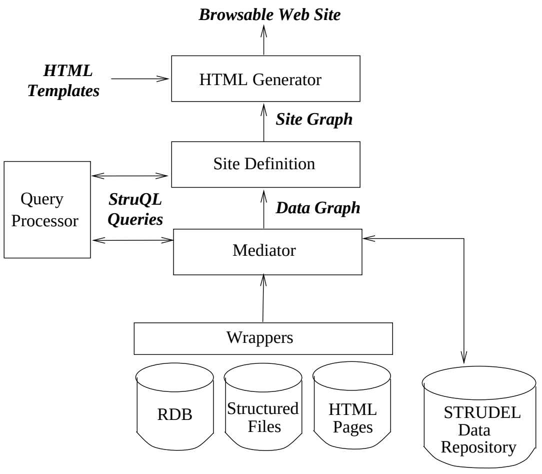

Figure 1: Strudel Architecture

2 The Strudel System

Strudel's architecture is depicted in Fig. 1; rectangles depict processes and emboldened terms specify the inputs and outputs of the processes.

2.1 System Architecture

Data model In every level of the Strudel system, the data model is a labeled, directed graph; this model is similar to OEM, which was developed in the TSIMMIS project [9]. The labeled graph model has been proposed for managing semistructured data, which often has few type constraints, a rapidly evolving schema, or missing schema.

In this model, the database consists of *objects* connected by directed edges labeled with string-valued *attribute names*. Objects are either nodes, identified by a unique object identifier (oid), or are atomic values, such as integers, strings, and files. Strudel supports several atomic types that commonly appear in Web pages e.g., URLs, and PostScript, text, image, and HTML files. The atomic types are handled in a uniform fashion, and values are coerced dynamically when they are compared at run time. Objects are grouped into named *collections*, which are used in queries. Objects may belong to multiple collections, and objects in the same collection may have different representations.

Data repository for semistructured data A Web site's data graph and site graph are stored in Strudel's data repository. The repository's initial data may be obtained from wrappers that convert data in external sources into an internal format. Data is exchanged between the data repository and external sources in a common data definition language, which in the style of OEM's data definition language [9].

STRUDEL's data repository, unlike those in traditional relational or object-oriented systems, can store data that lack schema information. Traditional systems rely on schema information to physically organize the data on disk, but our data repository cannot. Without schema information, we fully index both the schema and the data. For example, one index contains the names of all the collections and attributes in the graph; other indexes contain the extensions for each collection and attribute. In addition, indexes on atomic values are *global* to the graph, not built per collection or attribute. Obviously, maintaining these indexes is expensive, but they provide many benefits to our query language, which can also query the schema.

Mediator STRUDEL's mediator supports data integration by providing a uniform view of all underlying data, irrespective of where it is stored. When designing the mediator, we addressed two problems: whether to warehouse data from external sources or to access the external sources on demand at query time (see [20] for a comparison); and how to specify the relationship between the attributes and collections in the mediated schema and those in the data sources (see [24] for a discussion of possible approaches).

In STRUDEL's prototype, we implemented warehousing; the result of data integration is stored in STRUDEL's data repository. This simplified our implementation and sufficed for our applications, which have small databases. STRUDEL's architecture, however, can accommodate either approach.

Recent research addresses the problem of specifying the relationship between the mediated view of the data and the external data sources. *Global as view* (GAV) [2, 9, 17, 19, 23] and *Local as view* (LAV) [11, 18, 21] are two techniques. In GAV, the relationship between the two relations is specified by a set of queries. For each relation R in the mediated schema, a query over the source relations specifies how to obtain R's tuples from the sources. The LAV approach is the inverse: for every information source S, a query over the relations in the mediated schema describes how R's tuples can be found in S. GAV provides finer control over how to combine the data from the sources; in contrast, LAV simplifies adding and deleting sources and accommodates sources with overlapping data [24]. We found the GAV approach was suitable for STRUDEL, because it was immediately extensible to STRUQL[1] and because the number of data sources we integrated was small and did not change frequently, although the data in the sources may change frequently. Therefore, we did not have to change the mappings between the mediated schema and the source relations frequently.

Query processor STRUDEL provides a declarative language, STRUQL, for querying and restructuring semistructured data. Since both data graphs and site graphs are represented as labeled graphs, STRUQL queries can be applied to any graph, whether produced by a wrapper, a mediation query, or a site-definition query. As in traditional query processing, a query is first translated by the query optimizer into an efficient physical-operation tree. In STRUDEL's first implementation, we built a simple heuristic-based optimizer. Later, we developed a more comprehensive cost-based optimization algorithm [16]. The new optimizer can enumerate plans that exploit indexes on the data and the schema in order to choose the best plan. The optimizer is also well suited for accessing data in external sources when only limited access patterns are supported. Limited access patterns are common for semistructured-data sources, (e.g., they often require that some inputs be given to access the data) and pose novel challenges to query optimization. STRUDEL's query interpreter includes conventional physical operators as well as those necessary to query the schema (e.g., scan all the attribute names in a graph).

HTML generator To produce the HTML code for every page in the Web site, we associate an HTML template with every node in the site graph. HTML templates can be associated with collections of objects or with individual objects. Given an object and its HTML template, the HTML generator interprets the HTML template, replacing template expressions by the HTML values of the object's attributes. The resulting pages are the browsable HTML Web site.

2.2 The STRUQL Query Language

In STRUDEL, we need to query graphs and create new graphs at the *mediation* level, when data from different external sources is integrated into a data graph, and at the *site-definition* level, when site graphs are constructed from a data graph. We use a common query and transformation language, STRUQL (Site TRansformation Und Query Language) [13], at both levels. A query in STRUQL's core fragment has the form:

$$\begin{array}{ll} \textsf{where} & C_1, \ldots, C_k, \\ [\,\textsf{create} & N_1, \ldots, N_n] \\ [\,\textsf{link} & L_1, \ldots, L_p] \\ [\,\textsf{collect} & G_1, \ldots, G_q] \end{array}$$

A STRUQL query has two parts. The *query part* depends only on the where clause and produces all bindings of node and arc variables to values in the data graph that satisfy all conditions C_i in the where clause; its result is a relation with one attribute for each variable. The *construction* part (the create, link, and collect clauses) constructs a new graph from this relation to create nodes, arcs, and collections in the output graph. The result of the complete STRUQL query is a new graph.

For example, the following query returns all PostScript papers directly accessible from home pages:

$$\begin{array}{ll} \textsf{where} & HomePages(p), p \rightarrow \text{“}Paper\text{”} \rightarrow q, isPostScript(q) \\ \textsf{collect} & PostscriptPages(q) \end{array}$$

HomePages is a collection, "*Paper*" is an edge label, and *isPostScript* tests whether node q is a PostScript file. The distinction between collection names and external predicates is done at a semantic, not syntactic, level. 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.

In general, each condition $C_1, \ldots, C_k$ in a where clause either (1) tests collection membership, e.g., $HomePages(p)$, or (2) is a regular path expression, e.g., $p \rightarrow$ "*Paper*" $\rightarrow q$, or (3) is a built-in or external predicate applied to nodes or

[1] Extending the LAV approach to our context would require solving the problem of rewriting queries using views for the STRUQL language.

edges, e.g., $isPostScript(q)$. A condition of type (2) has the general form $x \to R \to y$ or $x \to L \to y$; the former means there exists a path from node x to node y that matches the regular path expression R, and the latter means there exists a single edge from node x to node y whose value is bound to the variable L. Regular path expressions are more general than regular expressions, because they permit predicates on edges and nodes. For example "$isName^*$" is a regular path expression denoting any sequence of labels such that each satisfies the $isName$ predicate. In particular, *true* denotes any edge label, and *true** any path; we abbreviate the latter with $*$. Other operators include path concatenation and alternation; the grammar for regular path expressions is: $R ::= Pred \mid (R.R) \mid (R|R) \mid R^*$.

The create and link clauses create new graphs from existing graphs. The following query produces a site graph called *TextOnly*, that excludes any nodes that contain image files:[2]

$$\begin{array}{ll} \mathsf{where} & Root(p),\ p \to * \to q,\ q \to l \to q', \\ & not(isImageFile(q')) \\ \mathsf{create} & New(p),\ New(q),\ New(q') \\ \mathsf{link} & New(q) \to l \to New(q'), \\ \mathsf{collect} & TextOnlyRoot(New(p)) \end{array}$$

New is a Skolem function that creates new object oids; by definition, a Skolem function applied to the same inputs produces the same node oid, so for some constant value of p, $New(p)$ always produces the same object. The query first finds all nodes q reachable from the root p (including p itself) and all nodes q' that are directly accessible from q by one link labeled l and that are not image files. For each node q and q', it constructs new nodes $New(q)$ and $New(q')$. This query effectively copies all nodes accessible from the root once. The query adds a link l between any pair of nodes that were linked in the original graph and adds a new link that points to the new root. Finally, it creates an output collection *TextOnlyRoot* that contains the new graph's root.

StruQL has an *active-domain* semantics and can be described in two stages, which correspond to the query and construction parts of a StruQL query. The meaning of the where-clause is a relation defined by the set of assignments from variables in the query to oid and label values in the data graph that satisfy all conditions in the where clause. The meaning of the create, link, collect clauses is as follows. For each row in the relation, first construct all new node oids, as specified in the create clause. Assuming the latter is create $N_1, \ldots, N_n$, each N_i is a Skolem function applied to node oids and/or label values. Next, construct the new edges, as described in the link clause. We require that each node in link or collect is either mentioned in create or is a node in the data graph. We also require that edges are added from new nodes to new or existing nodes; existing nodes are immutable and cannot be extended. Strategies for efficient evaluation and optimization of StruQL queries are described elsewhere [12].

```
collection Publications {
  abstract   text
  postscript ps
}
object pub1 in Publications {
  title  "Specifying Representations..."
  author "Norman Ramsey"
  author "Mary Fernandez"
  year   1997
  month  "May"
  journal  "Transactions on Programming..."
  pub-type "article"
  abstract   "abstracts/toplas97.txt"
  postscript "papers/toplas97.ps.gz"
  volume   "19 (3)"
  category "Architecture Specifications"
  category "Programming Languages"
}

object pub2 in Publications {
  title "Optimizing Regular..."
  author "Mary Fernandez"
  author "Dan Suciu"
  year   1998
  booktitle "Proc. of ICDE"
  pub-type  "inproceedings"
  abstract   "abstracts/icde98.txt"
  postscript "papers/icde98.ps.gz"
  category  "Semistructured Data"
  category  "Programming Languages"
}
```

Figure 2: Fragment of data graph for example site

2.3 Example Web Site

The following example shows how one author's homepage is generated by Strudel.[3] The main source of data for this homepage is the author's Bibtex bibliography file. The homepage site has four types of pages: the root page containing general information, an "abstracts" page containing all paper abstracts, "year" and "category" pages containing summaries of papers published in a particular year or category, respectively. We describe the first two steps of the site-definition process: creating the data graph from a Bibtex file and defining the site graph in StruQL.

Fig. 2 contains a fragment of the site's data graph and was generated by a Bibtex wrapper; the wrapper converts Bibtex files into a Strudel data graph. Both objects are members of the *Publications* collection. Because Strudel supports a semistructured data model, the names, types, and cardinality of attributes need not be identical. For example, *pub1* has a *month* attribute but *pub2* does not; *pub2* has a *booktitle* attribute, whereas *pub1* has a *journal* attribute. The *collection* directive specifies the default types of attribute values that would otherwise be interpreted as strings, e.g., *abstract* is a text file and *postscript* is a PostScript file. These directives are *not* constraints and can be overridden in the input file.

The site graph for the example homepage is defined by the query in Fig. 3. The first clause creates two new objects called *RootPage* and *AbstractsPage* and creates a link between them. The second clause (lines 7–8) creates two new objects, *AbstractPage(x)* and *PaperPresentation(x)* for each

[2] 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 only for the root page. Surprisingly, the following links point to pages with images.

[3] We encourage the reader to visit the Strudel-generated sites at http://www.research.att.com/~{mff,suciu} and http://www.cs.washington.edu/homes/alon/.

```
1     INPUT BIBTEX
2     // Create Root & Abstracts page and link them
3     CREATE RootPage(), AbstractsPage()
4     LINK RootPage()->"AbstractsPage"->AbstractsPage()
5
6     // Create a presentation for every publication x
7     WHERE  Publications(x), x->l->v         // Q1
8     CREATE PaperPresentation(x), AbstractPage(x)
9     LINK
10     AbstractPage(x) -> l -> v,
11     PaperPresentation(x) -> l -> v,
12     PaperPresentation(x)->"Abstract"->AbstractPage(x),
13     AbstractsPage() ->"Abstract" -> AbstractPage(x)
14
15     { // Create a page  for every year
16      WHERE  l = "year"                     // Q2
17      CREATE YearPage(v)
18      LINK
19       YearPage(v) -> "Year" -> v
20       YearPage(v)->"Paper"->PaperPresentation(x),
21
22       // Link root page to each year page
23       RootPage() -> "YearPage" -> YearPage(v)
24     }
25
26     { // Create a page  for every category
27      WHERE  l = "category"                 // Q3
28      CREATE CategoryPage(v)
29      LINK
30       CategoryPage(v) -> "Name" -> v,
31       CategoryPage(v)->"Paper"->PaperPresentation(x),
32
33       // Link root page to each category page
34       RootPage() -> "CategoryPage" -> CategoryPage(v)
35     }
36    OUTPUT HomePage
```

Figure 3: Site definition query for example homepage site

member x of the *Publications* collection; these presentation objects contain the publication's information that will appear in different parts of the site. For example, the expressions on lines 10–11 copy all of x's attributes and values into the new objects. The link clause also encodes inter-page structure. On line 13, the general abstracts page is linked to the abstract page of each publication (*AbstractPage(x)*). The nested where clause (lines 15–24) creates a page for each year associated with a publication; the link clause associates each *PaperPresentation* object with its corresponding *YearPage*. Lastly, the root page is linked to each year page. A similar clause creates a page for each publication category and links category pages to *PaperPresentation* objects.

Fig. 4 depicts a fragment of the generated site graph; for clarity, it excludes the result of the last nested clause that produces category pages. Note that the site graph encodes both the site's content and its structure. For example, the *PaperPresentation* objects have links to paper titles and to their associated abstract pages. All leaf objects contain page content, e.g., the titles of publications. Declarative specification of the site graph is powerful, because the site builder can specify its structure in any order he chooses. For example, he can define the pages "top down" from the root, or first define each group of related pages and then link them.

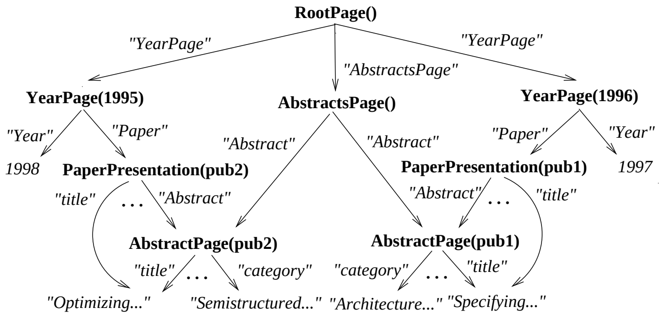

Figure 4: Fragment of site graph for example homepage site

2.4 HTML-Template Language

One premise of STRUDEL's design is that designing and *changing* the visual presentation of a site is separable from the management of the site's content and structure. Although HTML generation is not our central concern, we want STRUDEL users to be able to easily produce visually consistent and attractive sites. The result is a template language that extends plain HTML with three simple expressions: a format expression (SFMT), a conditional expression (SIF), and an enumeration expression (SFOR), each of which produces plain HTML text. This design evolved from our experience of representing sites in a semistructured data model. Fig. 5 contains the language's grammar.

The HTML generator takes as input a site graph and a set of HTML templates. For every internal object, the generator selects a HTML-template file for the object: either (1) an object-specific file, (2) the value of the object's HTML-template attribute, or (3) the template file associated with the collection to which the object belongs. Given an object and its HTML template, the HTML generator evaluates all expressions in the template, concatenates them together, and produces plain HTML text. It either emits the HTML value as a page or embeds the value in pages that refer to that object. The resulting pages contain the browsable Web site. Fig. 6 contains the HTML templates for the example site. The *RootPage*, *AbstractsPage*, *YearPage*, *CategoryPage* and *AbstractPage* are realized as pages.

The choice to realize internal objects as pages or as page components is delayed until HTML generation; our choice of Skolem function names (e.g., *AbstractsPage*) hints that some objects are realized as pages, but this is not required. For example, when referenced from the *PaperPresentation* template, an *AbstractPage* object is realized, by default, as a separate HTML page, but when referenced from the *AbstractsPage* template, the EMBED directive overrides this default and the *AbstractPage* object is embedded in the generated HTML page.

Associating an HTML template with a collection of objects allows the user to produce the same "look and feel" for related pages. Our technique for templating is much simpler than writing CGI programs to produce related pages; our plain template text is plain HTML with programmatic extensions, not a program that produces HTML text.

The template language provides three extensions to plain HTML. The format expression (<SFMT...>) maps an attribute expression into an HTML value. An attribute expression is either a single attribute, e.g., Paper, or a bounded sequence of attributes that reference reachable objects, e.g.,

```
Templ ⇒ {(Ext | Plain HTML Text)}

Ext ⇒ <SFMT AttrExpr [Format] >
    | <SFOR ID IN AttrExpr [Order] [Delim] > Templ </SFOR>
    | <SIF CondExpr > Templ [<SELSE> Templ] </SIF>

AttrExpr ⇒ [@] ID {. ID}

Format ⇒ (EMBED | [LINK [= Tag]])

Tag ⇒ {(STRING | AttrExpr)}

Order ⇒ ORDER=(ascend | descend) [KEY=AttrExpr]

Delim ⇒ DELIM=STRING

Expr ⇒ (AttrExpr| Constant)

CondExpr ⇒ Expr [Op Expr]
         | CondExpr (AND | OR) CondExpr
         | NOT CondExpr
         | ( CondExpr )

Constant ⇒ (BOOL | INT | FLOAT | STRING | NULL)

Op ⇒ (= | < | > | <= | >= | !=)
```

Figure 5: EBNF Grammar for HTML-Template Language

RootPage template:

```
<html><!-- Raw HTML text omitted -->
<h2><SFMT AbstractsPage LINK="All Paper Abstracts"></h2>
<h2> Publications by Year </h2>
<SFMT YearPage LINK=YearPage.Year UL
  ORDER=ascend KEY=Year><br>

<h2> Publications by Topic </h2>
<SFMT CategoryPage LINK=CategoryPage.Name UL
  ORDER=ascend KEY=Name>
<!-- More raw HTML text omitted --></html>
```

AbstractsPage template:

```
<html><H1>Paper Abstracts</H1>
<SFMT Abstract EMBED UL></html>
```

YearPage template:

```
<html><h1> Publications from <SFMT Year> </h1>
<SFMT Paper UL></html>
```

CategoryPage template:

```
<html><h1> Publications on <SFMT Name> </h1>
<SFMT Paper UL></html>
```

PaperPresentation template:

```
<SFMT postscript LINK=title>.
By <SFMT author ENUM DELIM=", ">.
<i> <SIF booktitle> <SFMT booktitle>
    <SELSE journal> <SFMT journal>
    </SIF>, <SFMT year>. </i>
<SFMT Abstract LINK="Abstract">
```

AbstractPage template:

```
<SFMT postscript LINK=title>.
By <SFMT author ENUM DELIM=", ">.
<i> <SIF booktitle> <SFMT booktitle>
    <SELSE journal> <SFMT journal>
    </SIF>, <SFMT year>.</i> <br>
<SFMT Abstract EMBED><br>
```

Figure 6: HTML Templates for example homepage site

`Paper.Name`. We found that limited traversal of the site graph is useful when writing HTML templates even though this feature overlaps with STRUQL's regular path expressions, which support both bounded and recursive traversal. Without limited traversal, the user is forced to encode all information to be displayed about an object in the object itself, which bloats site-definition queries and destroys their modularity.

Format expressions are concise, because the HTML generator uses type-specific rules to determine an attribute's HTML value. For most atomic values (integers, strings, URLs, HTML and text files), the attribute's HTML value is converted to a string and is embedded in the HTML template. For example, in the *YearPage* template, `<SFMT Year>` is replaced by the object's `Year` attribute, which is an integer. Some values, such as PostScript files, should not be realized as strings. For these values, the HTML generator produces an appropriate link to the value. For example, in the *PaperPresentation* template, `<SFMT postscript LINK=title>` is replaced by a link to the object's `postscript` attribute, which is a PostScript file, and the object's `title` attribute is emitted as the link tag. When an attribute expression refers to an internal object, the HTML generator replaces the expression with the object's HTML value; if the internal object is a page, a link to its HTML file is emitted, otherwise its HTML value is embedded in the current page.

Because the semistructured data model permits objects in the same collection to have different representations, it is often necessary to test for the existence of an object's attribute or test its value in a template. The expression `SIF` evaluates a condition and if it is true, evaluates the first template expression, otherwise it evaluates the optional second expression. A condition can test whether an attribute expression is non-null and can apply relational operators to attribute expressions and constants. In the *PaperPresentation* template, for example, attributes common to all objects in the collection (e.g., `author`, `postscript`, `year`, and `Abstract`) are emitted directly. The object-specific attributes (`booktitle` and `journal`) are emitted conditionally.

The semistructured data model also permits an object to have multiple instances of the same attribute, e.g., `RootPage` has multiple `YearPage` attributes. The same attribute can refer to object values of different types. The iteration expression `SFOR` iterates over all values of the attribute expression, binds the variable *ID* to each value, and evaluates the nested template expression for each binding. The variable may reference an internal object or an atomic value. If it references an internal object, it may be used in attribute expressions. For example, the following expression binds a to every value of the attribute author and embeds each of a's values: <SFOR a IN author DELIM=","><SFMT @a EMBED></SFOR>.

Enumerating all values of an object's attribute is common, so we abbreviate common idioms. For example the expression <SFMT author ENUM DELIM=","> in the *PaperPresentation* template is equivalent to the expression above. Attributes are often emitted in ordered and unordered lists. For example, <SFMT Abstract EMBED UL> in the *AbstractsPage* template is shorthand for:

```
<UL>
<SFOR a IN Abstract><LI><SFMT @a EMBED></SFOR>
</UL>
```

It is not possible to specify order of attributes in our data model; however, we often want to display attributes in a specific order. The ORDER directive sorts an attribute's values in either lexicographically increasing or decreasing order; if the attribute's values are internal objects, the optional KEY value specifies the object's attribute that should be used as the key. For example, the expression <SFMT YearPage UL ORDER=ascend KEY=Year> in the *RootPage* template sorts all the YearPage values in ascending order, uses their Year values as a key, and emits them in an unordered list.

2.5 Site Schemas

StruQL permits the site builder to construct fragments of a Web site separately using multiple queries. To view a site's complete, abstract structure, we generate a *site schema* from the site's StruQL queries; site schemas are a refinement of graph schema [7]. Because StruQL's query and construction stages are separate, a simple analysis of the query can infer the *site schema* of the site graph. Given a query Q, a site schema is an equivalent reformulation of Q in terms of a graph, which specifies the *possible* paths in a Web site generated by Q. Formally, the site schema for a query Q is a labeled graph G_Q. G_Q has one node N_F for every Skolem function symbol F in the query, plus one additional special node, NS, corresponding to non-Skolem nodes in the site graph. The graph has one edge $N_F \rightarrow N_G$ for every link expression $F(\bar{X}) \rightarrow L \rightarrow G(\bar{Y})$ in Q, and one edge $N_F \rightarrow NS$ for every link expression $F(\bar{X}) \rightarrow L \rightarrow V$, where V is a variable. Each edge in G_Q corresponding to a link expression $F(\bar{X}) \rightarrow L \rightarrow G(\bar{Y})$ is labeled $(Q, L, \bar{X}, \bar{Y})$, where Q is the query in the where-clause associated with that link expression; each edge corresponding to a link expression $F(\bar{X}) \rightarrow L \rightarrow V$ is labeled $(Q, L, \bar{X}, [V])$.

The site schema is equivalent to the original query, i.e., we can recover the query from the site schema. Fig. 7 depicts site schema that corresponds to the query in Fig. 3; for clarity, edges to the NS node are excluded. The link expression YearPage(v) -> "Paper" -> PaperPresentation corresponds to the edge *YearPage* → *PaperPresentation* labeled $(Q1 \wedge Q2, \text{“}Paper\text{”}, [v], [x])$. Note that the query that governs creation of this link is the conjunction of the where clauses $Q1$ and $Q2$.

We use site schemas in several ways. Site schemas serve as a visual summary of the site graph, which is valuable during the iterative definition of a Web site's structure and allows visual verification of a site's integrity constraints (e.g., connectedness, reachability of nodes.) They are also the basis for an algorithm that enforces integrity constraints on Strudel-generated sites [14] and for an incremental-evaluation algorithm that converts one site-definition query into multiple queries that are evaluated dynamically when a user browses the site [15]. We briefly outline their application to these problems.

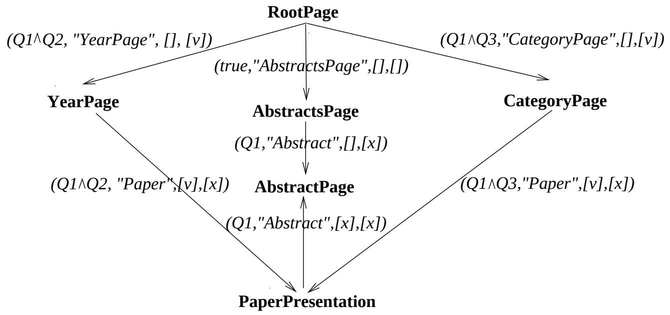

Figure 7: Site schema for query in Fig. 3

Verification of integrity constraints Visual verification is adequate for simple constraints, but verifying arbitrary integrity constraints requires automation. We often want to enforce constraints that refer to the site graph, e.g., "All paper presentation pages are reachable from a category page". We define the verification problem as an entailment problem of a StruQL query and a logical sentence describing the integrity constraint. Integrity constraints are logical sentences built from expressions of the form $C(X)$ and $X \rightarrow R \rightarrow Y$ using logical connectives and quantifiers, e.g., the constraint above is expressed by $(\forall X) PaperPresentation(X) \Rightarrow CategoryPage() \rightarrow * \rightarrow X$. In general, constraint formulae can refer to the data and site graphs, but we can only reason about formulae that refer to the data graph. Site schemas allow us translate constraint formulae on the site graph into formulae on the data graph.

Dynamic computation of site graph In Strudel's prototype, StruQL queries are evaluated statically, i.e., complete site graphs are computed before users browse them. This approach is feasible for sites whose data changes infrequently, but it is infeasible for sites that are updated frequently. Moreover, completely materializing a site graph is often impossible, for example, when pages depend on user input derived from forms. In these cases, some nodes in a site graph must be created dynamically. To dynamically create a site, Strudel must evaluate at "click time" the incremental query that computes the data required to display the next page. Site schemas specify, for each node in the site graph, the queries that must be evaluated to compute the node's contents, i.e., its outgoing edges. Naive evaluation of these queries is costly, because they often recompute information derived for already browsed pages; consider, for example, all the edges in Fig. 7 labeled $Q1 \wedge Q2$. For a particular node, we can optimize its incremental query using contexts derived from the paths that reach the node and also precompute "lookahead" results for queries of reachable nodes. Site schemas specify the queries for paths that begin and terminate at nodes in a site graph.

3 Related Systems

Strudel is one of several systems that support restructuring and creation of Web sites. Whereas Strudel's focus is on integrating data from various types of data sources and on generating new sites, other systems focus on extracting structure from existing Web pages and on producing a new view. Both the WebOQL [3] and Araneus [5] systems support querying of existing Web sites and can produce views of sites as restructured graphs. WebOQL is similar in spirit to STRUDEL. Like STRUDEL, WebOQL provides a uniform, semistructured data model (called a *hypertree*), and its query language supports regular path expressions, can restructure graphs, and is compositional; unlike STRUDEL, its data model supports records and ordering. Also, WebOQL expresses the HTML rendering of pages in queries.

Like WebOQL, Araneus converts existing Web pages into an abstract data model (*ADM*), which is a graph of strictly typed *page schemes* that specify the content of related pages. Araneus provides a query language (ULIXES) for defining a relational view of an ADM graph; multiple data sources are integrated by relational queries over these relational views. A second query language (PENELOPE) transforms an integrated, relational view back into an ADM graph; a final step renders an ADM graph as a browsable site. Like STRUDEL, Araneus separates data integration, site definition, and visual presentation, but it requires two data models, its page schemes must be specified explicitly, and its two query languages cannot be composed naturally.

The Autoweb [22] system is based on the hypermedia design model (HDM), a design tool for hypermedia applications. Like STRUDEL and Araneus, Autoweb separates site-management tasks: the "hyperbase schema" describes the site's content in HDM, which is based on the entity relationship model; the "access schema" specifies how the hyperbase is navigated and accessed in a browsable site; and the "presentation schema" specifies how objects and paths in the hyperbase and access schemas are rendered. Although the hyperbase and access schema are distinct, the navigation paths in an access schema are tightly coupled to the entity relationships in a hyperbase schema. Autoweb does not support querying or data integration.

4 Questions Answered in Our Study

After implementing STRUDEL's first prototype, we wanted to evaluate STRUDEL's methodology and our choice of the semistructured data model. First, we considered whether our premise that the three tasks of Web-site creation can and should be separated holds in practice. Specifically,

- Is there always a clear separation between these tasks? If not, in which cases do their mutual dependencies make separating them counter productive?
- For what kinds of Web sites is STRUDEL most effective? How useful is the ability to explicitly and declaratively manage a Web site's structure?

Regarding STRUDEL's data model and STRUQL's support for querying semistructured data, we asked:

- What characteristics of semistructured data were most important in STRUDEL? Conversely, why could we not effectively implement STRUDEL in a traditional data model?
- Are STRUQL's features, e.g., regular path expressions and restructuring capabilities, necessary for site definition? What features are missing from STRUQL that might simplify site definition?

5 Experiences with STRUDEL

We have had both practical and exploratory experiences with STRUDEL. In our practical experience, we used the STRUDEL prototype to create sites for individual users and for two organizations and to create a version of the CNN Web site for demonstration purposes. In our exploratory experience, we described our methodology and demonstrated our prototype to several potential commercial users. We describe this experience, addressing the above questions wherever relevant.

5.1 Practical experience

Our largest examples to date are the internal and external Web sites of AT&T Labs–Research. We built versions of these sites that are identical to those built by our Web masters [4]. This site is typical of an organization's site: it includes home pages of individual members, pages on projects, demos, research areas, and technical publications. The internal site is similar to the external site, but includes organizational and proprietary information. The data sources for this site are small relational databases that contain personnel and organizational data, structured files that contain project data, and existing HTML files. The wrappers are simple AWK programs that map structured files and relational databases into objects in a data graph. The wrappers for plain HTML pages are hand written.

The internal site generated by STRUDEL contains the home pages of approximately 400 users and pages for organizations and projects. The internal site is defined by a 115-line query and 17 HTML templates (380 lines). STRUDEL's power is revealed in the definition of the external site: no new queries were written for that site. Both the internal and external sites share the same site graph and many HTML template files. Only five HTML template files differ for the external site and these either exclude or reformat information that cannot be viewed externally.

Our own home pages are examples of small sites generated by STRUDEL[5]. The main data sources for these sites are our bibliographies. A simple wrapper maps BIBTEX files into data graphs; other information is stored in files in STRUDEL's data definition language. The `mff` example shows how to generate internal and external versions of the same homepage. The example contains data from two sources: a Bibtex file and a STRUDEL data file, which contains personal information, such as address, phone, projects, professional activities, patents, and is defined by a 48-line query and thirteen HTML templates (202 lines). The HTML templates for the external version exclude patents, and any publications and projects that are proprietary. The `suciu` example illustrates how to integrate data from multiple sources. Its site graph is built in several successive steps by multiple, composed STRUQL queries; for example, the last step copies the entire site graph and adds a navigation bar to each page. This example also shows how to define multiple views with STRUQL instead of with HTML templates.

We are also working on a STRUDEL-generated version of the INRIA-Rodin Web site, which is similar to the AT&T Research site. Its main feature is that the site has two views: one English and one French. The two sites are cross-linked

[4] The official external site is at `http://www.research.att.com` and is generated using a large set of CGI-BIN scripts.

[5] See `http://www.research.att.com/{~mff,~levy,~suciu}`.

so that each English page is linked to the equivalent page in the French site and vice versa. One StruQL query defines both views and creates the links between them.

Our first example was a demonstration version of the CNN Web site (`http://www.cnn.com`). On any day, one article may appear in various formats on multiple pages in the CNN site. Because we did not have access to CNN's databases of articles, we mapped their HTML pages into a data graph containing about 300 articles. Our version of the CNN site is defined by a 44-line query and nine templates. To demonstrate Strudel's ability to generate multiple sites from one database, we also generated a "sports only" site that has the same structure as the general site, but contains articles on sports subjects[6]. The sports-only query is derived from the original query and only differs in two extra predicates in one where clause. Both sites use the same templates.

5.2 Exploratory experience

We have described our methodology and demonstrated our prototype to several potential users, including the Web-site managers for AT&T's internal organizations, the Web-site management team at CNN, a company that designs and publishes sites, and a company that creates Web-based interfaces and data-integration technologies for business applications. Each group identified benefits of using Strudel that we did not anticipate and problems that must be addressed in an industrial quality implementation.

One unanticipated but important benefit is that Strudel could be used to generate sites tailored to individual users. CNN currently provides a custom-news site; a user selects those news categories that he would like in his personal site, and the server generates pages that contain articles from those categories. The user has no control over the generated site's structure. A custom StruQL query would allow the user to organize his news as he wanted and allow CNN to generate pages that contain advertisements targeted to that user. Strudel's separation of site management and visual presentation make this feasible. Another application of user-specific sites is producing custom interfaces for different types of users (e.g., marketing, customer care, analyst) that require access to the same databases, but that want to view the information in different ways.

Potential users uniformly agreed that the ability to integrate information from multiple sources while building a Web site is valuable. They also agreed that managing the structure of Web sites is a problem of growing importance. Both the CNN team and Web-site design firm indicated, however, that they would need to edit both the structure and content of the generated pages and that these changes should be propagated automatically back into the HTML templates, site-definition query, or underlying data. Several customers noted that a graphical interface for specifying StruQL queries in the spirit of Query By Example [26] would be necessary.

6 Evaluation

We describe the lessons we learned from our experience using Strudel and evaluate its methodology, its query language, and its semistructured data model.

6.1 The Strudel Methodology

Separating the management of the underlying data from other Web-management tasks is the basis for several commercial products, e.g., most commercial relational and object-oriented databases provide Web interfaces to their systems. Strudel provides two other important features: the abilities to integrate data from *multiple* sources and to incorporate unstructured sources (e.g., structured files). The AT&T Research site, for example, integrated five data sources.

Isolating the management of a site's structure was also important. For example, CNN's Web-site group is building a specialized tool for managing site structure. We also found that building complex Web sites is an iterative process in which the site structure evolves over time. For example, creating the AT&T and Rodin sites required several iterations. Declarative specification of the site's structure enables easy changes to a site. Finally, Strudel is most effective when multiple versions of a site are built from the same underlying data. For instance, once we built AT&T's internal research site, building the external version was trivial.

Separating management of the site's structure and its visual presentation is more subtle. This separation simplifies creating multiple versions of a site especially when the site's structure is the same in all versions, but its visual presentations differ. In this case, all versions share one site graph, but each version has its own HTML templates. It is not always clear, however, which aspects of a site should be encoded as structure or as visual presentation. For example, the AT&T external site is derived from the internal site by excluding the attributes of some objects in the generated pages; in this case, it is easier to create HTML templates that omit these attributes than it is to create a new site graph that explicitly excludes those attributes. Consider the order of articles or the placement of images in a page at the CNN site. Such information could be encoded in the visual presentation or in the site's structure. For CNN, managing this information is crucial, because they consider these editorial elements a primary value of their site.

To characterize the sites for which Strudel is most useful, we consider two criteria: the amount of data they contain and their structural complexity (see Fig. 8). Measuring the amount of data in a site is straightforward. One possible measure of structural complexity is the number of link clauses in the site-definition query. In current practice, an analagous measure of site complexity is the number of CGI-BIN scripts required to generate a site.

We observed that Strudel is most useful for sites that have complex structure and whose structure is dependent on the underlying data. For example, the CNN Web site contains a large number of articles. Although the disposition of an article in a site is complex (i.e., it appears in several formats on different pages and is linked to many other pages), the structure is uniform for all articles in the site. This uniformity also applies to all people in the AT&T site and all publications in the example homepage sites.

Fig. 8 categorizes the suitability of different Web-creation tools for various kinds of sites. When a site has simple structure and little data (lower left), WSYWIG tools such as Microsoft FrontPage or NetObjects Fusion are appropriate. When a site contains large amounts of data, but has simple structure, then a tool that provides a Web-based interface to a database is appropriate. When the data is large and the site structure is complex, Strudel is most appropriate.

[6] The versions of the CNN general and sports-only sites are at `http://www.research.att.com/~mff/presentation/strudel-demo.html`.

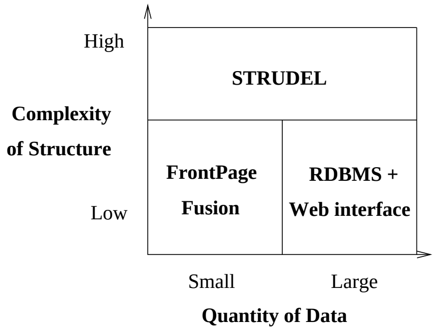

Figure 8: Suitability of Web-site management tools

6.2 StruQL Query Language

We were surprised by how well StruQL was suited for the application. StruQL's most important feature is separation of the query (where) and construction (create, link, and collect) stages. This separation is natural, because it is clear conceptually to separate extraction of data from external sources and site construction. Another benefit is that the query stage is independently extensible; for example, we could extend it to include grouping and aggregation. This separation also simplifies query optimization, because all where clauses can be evaluated by an optimizer at once. It should be noted that other languages (e.g., WebOQL [3], UnQL [8]) do not separate selection and restructuring.

StruQL's declarative semantics were also important. A site builder often designs related parts of a site's structure individually then links them together. The ability to specify link clauses in whatever order is natural makes this possible. More importantly, StruQL's declarative semantics allow us to perform more complex site-management tasks such as guaranteeing that integrity constraints are satisfied by Strudel-generated sites and automatically converting a complete site-definition query into multiple queries that can be evaluated statically or dynamically at "click time".

We found that the StruQL queries for sites with complex structure tend to be long. To simplify writing queries, we introduced nested queries, and we allowed queries to *add* nodes and arcs to a graph, instead of creating a new graph in every query. This allows different queries to create different parts of the same site. Finally, we built a tool to view a query's site schema, which provides a visual map of the site being specified.

Arc variables, which are bound to arc labels in the graph and, hence, to elements of the graph's schema, were an important feature, because they *carry over* irregularites in the data to the site graph. In the example in Fig. 3, the expression on line 11 copies every attribute of a *Publication* object into a *PaperPresentation* object; this set of attributes will differ for each *Publication* object.

Because our applications' data sources were mostly structured, the site-definition queries rarely used the Kleene star in StruQL's regular path expressions. Regular path expressions are useful when the possible sequences of attribute names in the data are not known in advance. This did not occur in our examples. Regular path expressions are useful in other applications of StruQL. For example, they can express integrity constraints on a site graph, e.g., "all pages are reachable from the site's root" or "every department member is reachable from a department page", and they may be useful for querying a Strudel-generated site.

6.3 Semistructured Data

Our experience indicates that the semistructured model was the right choice for Strudel. We describe its most important features, in particular, the ability to easily evolve the schema and to manage irregular data.

The semistructured data model does not require that the schema be defined before the data, which simplifies modifying the schema. This was an important feature in Strudel. Creating both the data and site graphs was an iterative process, so the ability to modify their schema was important. We first wrote wrappers for the external data sources and generated the integrated data graph; then we wrote a site-definition query, applied it to the data graph, and generated the site graph. We repeated this process until we discovered all the information from the external sources that we wanted displayed in the site. For example, the AT&T data graph integrated data from several structured and unstructured sources. While writing the wrappers for these sources, the data graph's schema changed frequently, e.g., several attributes were added to the schema on-the-fly.

Defining a site graph requires even more flexibility, because its structure is not defined explicitly by a schema, but implicitly by a StruQL query. Even after the site is constructed, we often want to change its structure, and therefore its schema as well. During definition of the AT&T site, for example, we discovered similarities between pages that were not explicit in the site graph. The information about lab and department directors initially was modeled by two different collections; over time, we discovered that objects in these collections shared many common attributes, so we merged the two collections. Because of the dynamic nature of site and data graph schemas, we conclude that traditional relational and object-oriented model are not appropriate.

In Strudel, we associate sets of objects with *collections*. A collection is like a class, except that objects need not have identical representations, i.e., the same attributes or the same attribute types. This model supports irregular structure in the data. We encountered several kinds of irregularity in our data, such as missing or extra attributes. There are several sources of such irregularities. First, attribute values may be missing because they were omitted during data entry. In the AT&T site, for example, some projects omitted the "synopsis" attribute. Second, no values may exist for some attributes at a given time. Not all projects in AT&T are sponsored, and therefore have no value for the "sponsor" attribute. Third, some attributes are not meaningful for certain objects. In the *Publications* collection, the "journal" attribute is meaningful for journal papers, but not conference papers. Finally, even when objects have the same attribute, they may not be of the same type. For example, an address may be a string in one object, but a structure with address, city and zipcode fields in another object. Although we did not encounter this irregularity in our examples, we expect that such irregularity will arise for sites that integrate overlapping data from multiple sources.

Modeling irregular data in an object-oriented model would require either building an artificial class hierarchy (where each class had exactly the same set of attributes), or constructing a maximal schema, where each object has all attributes. Furthermore, handling attribute values of different types would be cumbersome.

A recurrent issue was how much structure *should* be provided by a semistructured data model. In our initial design, we found that we needed collections, but we did not anticipate the need for ordered lists. For example, objects in the *Publications* collection have an associated *list* of authors. Maintaining order among authors is necessary when displaying the object in a Web page. Supporting lists in the data model, however, increases the complexity of query evaluation and optimization. Instead, we developed a solution (associating an integer key with each author) that allows us to preserve order in specific, but common, cases.

7 Future Research

Our experience helped us identify research problems of practical and theoretical interest. They address issues of STRUDEL's applicability to dynamically generated sites, its scalability to larger sites, its usability as an end-user tool, and its interoperability with existing tools.

In STRUDEL's prototype, we precompute a Web site by completely materializing its site graph. Most Web sites, however, cannot be precomputed, because they depend on user input that is not available statically or because the underlying data sources are too large. Currently, STRUDEL does not support dynamically generated sites. In practice, dynamic generation is supported by often large sets of loosely related CGI programs. Supporting dynamic evaluation would eliminate writing such programs by hand.

Although we can decompose a site-definition query into multiple, dynamic queries, and we have theoretical techniques for optimizing these queries, implementing dynamic evaluation requires significant systems-design effort. For example, our optimization techniques cache query results to reduce "click time" for future queries; these results essentially encode state required by the STRUDEL query processor. An open problem is how and where this state should be stored: in a client-side browser and/or a server-side query processor. To solve this problem, we expect to use existing systems and techniques that support stateful Web services [4].

Although adequate for a prototype, STRUDEL's warehousing mediator is inadequate for sites whose data sources are large or change frequently. To support large-scale sites, we need to solve the problem of incremental view updates for semistructured data, which is an open problem. An alternative approach is translating a query on a mediated schema into a set of queries on the relevant data sources. Although this problem has been addressed for (unions of) conjunctive queries and some forms of recursive queries, it has not been addressed for languages over semistructured data. In particular, arc variables, i.e., querying the schema, and the restructuring operators create and link introduce difficulties.

Traditional database systems rely heavily on schema information to organize data on disk. An important problem is developing analogous techniques for semistructured data in which schema information is missing or changes frequently. This problem is related to the problem of dynamic object reclustering in object-oriented databases. Traditional systems use query patterns to choose indexes to build. In STRUDEL, however, identifying query patterns is complicated by STRUQL's features that permit querying schema.

Not surprisingly, many potential users of STRUDEL asked whether we can provide a friendly visual interface for specifying queries, instead of having to write STRUQL queries by hand. Clearly, a better interface is needed, probably in the spirit of Query By Example [26]. One research issue is what subset of STRUQL can be expressed using a graphical interface. A similar issue has arisen for other graphical query languages such as Hy$^+$ [25].

Many commercial tools exist for Web-site creation and management. We do not presume that STRUDEL will replace *all* of them, therefore an important practical issue is how to integrate STRUDEL with existing tools. Developing the appropriate API to STRUDEL may be the best way to incorporate it into tools that Web-site builders currently use.

8 Conclusions

This work makes several important conceptual and practical contributions. First, we identified Web-site creation and management as data-management problems that can benefit from database technologies, and in particular, benefit from declarative specification of a site's content and structure. We also recognized that separating the management of a site's data, the management of its structure, and the visual presentation of its pages, facilitates many site-management tasks, such as integrating data from multiple sources, generating multiple views of a site, modifying a site's structure over time, and enforcing integrity constraints on sites.

Second, we identified STRUDEL as an ideal application of the semistructured data model, because that model supports data integration and can handle data with irregular structure and rapidly evolving schema. We also provided a detailed description of the characteristics of semistructured data that were most relevant to our application and explain why traditional models proved inadequate.

Third, we built a prototype of STRUDEL that supports the semistructured data model and provides a query processor for STRUQL, which handles graph querying and restructuring. This required us to solve several technical problems, such as designing a data repository for semistructured data and designing optimization algorithms for queries over semistructured data. We also developed a simple yet powerful HTML-template language that supports HTML presentation of objects in our data model. Our prototype also provides an implementation platform for future research on semistructured data and query optimization.

Finally, our experience using STRUDEL to build several Web sites validated our key assumptions that separation of the three site-management tasks is natural in practice and that declarative specification of site content and structure effectively supports the tasks described above. Our experience also identified the problems that would have to be solved in a production quality implementation of STRUDEL and that require additional research. The practical problems include designing a graphical user interface to STRUQL and integrating STRUDEL's functionalities with existing Web-management tools. Other problems, such as computing incremental updates of site graphs, decomposing queries to support dynamic computation of sites, and designing efficient storage representations for semistructured data, have broader implications in the field of semistructured data and pose harder challenges. We have already begun addressing these problems and plan to investigate practical solutions in future versions of STRUDEL.

References

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the ICDT*, 1997.

[2] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of SIGMOD-96*, 1996.

[3] G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, database and webs. In *Proceedings of International Conference on Data Engineering*, pages 24–33, 1998.

[4] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proceedings of Conference on Domain-Specific Languages*, pages 37–49, 1998.

[5] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *Proceedings of VLDB*, pages 206–215, 1997.

[6] P. Buneman. Semistructured data. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona*, pages 117–121, 1997.

[7] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[8] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of SIGMOD–96*, pages 505–516, 1996.

[9] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources. In proceedings of IPSJ, Tokyo, Japan, October 1994.

[10] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion. In *To appear in Proceedings of SIGMOD*, 1998.

[11] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona.*, 1997.

[12] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. System demonstration - STRUDEL: A web-site management system. In *ACM SIGMOD Conference on Management of Data*, 1997.

[13] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[14] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Reasoning about Web-site structure, 1998. Submitted for publication.

[15] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Warehousing and incremental evaluation for Web-site management, 1998. Submitted for publication.

[16] D. Florescu, A. Levy, and D. Suciu. A query optimization algorithm for semistructured data. Technical report, AT&T Labs, 1997.

[17] D. Florescu, L. Raschid, and P. Valduriez. A methodology for query reformulation in CIS using semantic knowledge. *Int. Journal of Intelligent & Cooperative Information Systems, special issue on Formal Methods in Cooperative Information Systems*, 5(4), 1996.

[18] M. Friedman and D. Weld. Efficient execution of information gathering plans. In *Proceedings of IJCAI*, 1997.

[19] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd VLDB Conference, Athens, Greece*, 1997.

[20] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona*, pages 51–61, 1997.

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

[22] P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *Proceedings of EDBT Conference*, Valencia, Spain, 1998.

[23] A. Tomasic, L. Raschid, and P. Valduriez. A data model and query processing techniques for scaling access to distributed heterogeneous databases in Disco. *IEEE Transactions on Computers, special issue on Distributed Computing Systems*, 1997.

[24] J. D. Ullman. Information integration using logical views. In *Proceedings of the International Conference on Database Theory*, 1997.

[25] P. 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.

[26] M. Zloof. Query-by-Example: a data base language. *IBM Systems Journal*, 16:4:324–343, 1977.