Submission No. 272

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

Mary Fernandez† Daniela Florescu Jaewoo Kang Alon Levy
Dan Suciu
AT&T Labs, Florham Park, NJ
{mff,dana,kang,levy,suciu}@research.att.com

November 3, 1997

Abstract

Managing Web sites is a tedious data-management task, because today's Web sites often are derived from multiple data sources and have complex structures. To simplify these tasks, Web-site builders need tools for managing Web sites' content and structure.

The Strudel system applies concepts from database management systems to the process of building Web sites. The key idea is separating the management of the site's data, the creation and management of the site's structure, and the graphical presentation of the site's pages. First, the site builder creates a uniform model of all the data available at the site. Second, the site 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 site builder specifies the graphical 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. In particular, we describe our experience constructing several Web sites with Strudel and discuss the impact of potential users' requirements on Strudel's design. We address several specific questions: (1) when does a declarative specification of site structure provide significant benefits, (2) what are the main advantages provided by the semi-structured data model, and (3) what techniques for data integration are most appropriate for a system like Strudel.

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 the tasks that a Web-site builder must address: (1) choosing and accessing the data that will be displayed at the site, (2) designing the site's structure, i.e.,

*This title alludes to a well-known statement that the database community has "missed the boat" with respect to the Internet.

†Contact author: Mary Fernandez, AT&T Labs - Research, 180 Park Ave., Bldg 103, B291, Florham Park, NJ 07932-0971, 973-360-8679, (FAX) 973-360-8187, mff@research.att.com

specifying the data contained within each page and the links between pages, and (3) designing the graphical 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 graphical 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. The desire to support these tasks naturally leads us to view the problem from a data management perspective.

We have developed the STRUDEL system (demonstrated at SIGMOD-97 [9]), which applies concepts from database management systems to Web-site creation and management. In particular, STRUDEL illustrates the value of declarative specification of the content and the structure of Web-sites and the use of declarative query languages to automatically define Web sites. STRUDEL's key idea is separating the management of a Web site's data, the management of the site's structure, and the graphical presentation of the site's pages.

Using STRUDEL, the site builder first defines 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. The first step in building a site is to create an integrated view of the data underlying the site. This is done by STRUDEL's mediator component. In the mediator level as in all other levels of the system, all data, whether stored externally or internally, is modeled as a labeled directed graph. 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. Both the site and data graphs are represented in a common data model of labeled directed graphs, which is the model commonly used for semistructured data [1, 3]. Third, the builder specifies the graphical 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 provides several benefits. Since a Web site's structure and content are defined declaratively by a query, not procedurally by a program, it is very easy to create multiple *versions* of a site, e.g., internal and external views of an organization's site or 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 graphical presentations for the same site graph. The evolution of a Web site's structure is also facilitated by STRUDEL's architecture. For example, if we want to reorganize information in pages based on frequent usage patterns or to extend the site's content, we simply rewrite the site-definition query. We expect that declarative specification of Web sites will offer more advantages in the future. In particular, it becomes possible to express and enforce integrity constraints on the site and to update a site automatically when relevant changes occur in the underlying data.

STRUDEL is the first system that clearly separates the three tasks of building Web sites. Some existing tools support one or two of these tasks, but none address all three or support their complete separation. Some tools (e.g., Vignette's StoryServer, tools provided by major database vendors) separate the management of the underlying data from its graphical 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 paper reports on our experiences using STRUDEL. We first describe STRUDEL's architecture and the main design choices we made and present the data management problems that arise when building complex Web sites. We describe our experience constructing several Web sites with STRUDEL and discuss the impact of potential users' requirements on STRUDEL's design. Finally, we highlight the research problems that arose from our experience. Our study addresses two main questions:

- STRUDEL's design separates the three tasks of Web-site creation, which facilitates the declarative definition of a site's structure. We discuss whether this separation is natural in practice and under what circumstances declarative specification of site structure provides significant benefits. Our experience shows that management of site structure is becoming a central problem in managing complex Web sites. Isolating the management of a site's underlying data is also an important design decision. We observed that in many cases, a site's structure can be encoded either in the site graph or in the graphical presentation. This led us to develop a powerful HTML-template language whose functionality overlaps the STRUQL query language. This overlap permits users to encode their sites in whatever way they find most natural.

- 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, 3]. This data model was appealing because Web sites are graphs with irregular structure and non-traditional schemas. Furthermore, semistructured data facilitates integration of data from multiple, non-traditional sources. We discuss the suitability of this data model for our application. Specifically, we identify which characteristics of semistructured data were most important in STRUDEL and what prevented us from using a traditional data model. We also examine whether STRUQL, STRUDEL's query language for semistructured data, can naturally specify the structure of Web sites and consider the tradeoffs of adding various types of structure to the data model. We observed that the greatest benefits of a semistructured data model are that it allows the Web site's schema to evolve gradually and that it permits objects belonging to the same class to have multiple representations. This experience indicates STRUDEL is an appropriate application of semistructured data.

We also describe two technical problems that arose from our experience and that we solved in STRUDEL. In particular, we describe STRUDEL's HTML template language that was developed when we observed that the separation between site structure and graphical presentation is often unclear. We also describe the method we developed to generate *graph schemas* for STRUQL queries. Graph schemas allow the user to view a skeleton of the site's structure during site design. As stated above, STRUDEL includes a data-integration module that provides the Web site builder with a uniform view of the site's data. Several data-integration techniques have been proposed recently, but there is little experience comparing their strengths. We discuss which approach is most appropriate for STRUDEL.

In Section 2, we describe the different components of the STRUDEL system, including the novel features added as a result of our experience. Section 3 outlines the questions we address in our study. Section 4 describes the sources of our experience, and Section 5 discusses the lessons we learned from our experience.

2 The STRUDEL System

In this section, we describe STRUDEL's architecture and its different components. STRUDEL's architecture is depicted in Figure 1; rectangles depict processes and emboldened terms specify the inputs and outputs of the processes. We then describe in more detail STRUDEL's query language and HTML-template language. Section 2.3 provides a complete example of constructing a simple Web site using STRUDEL.

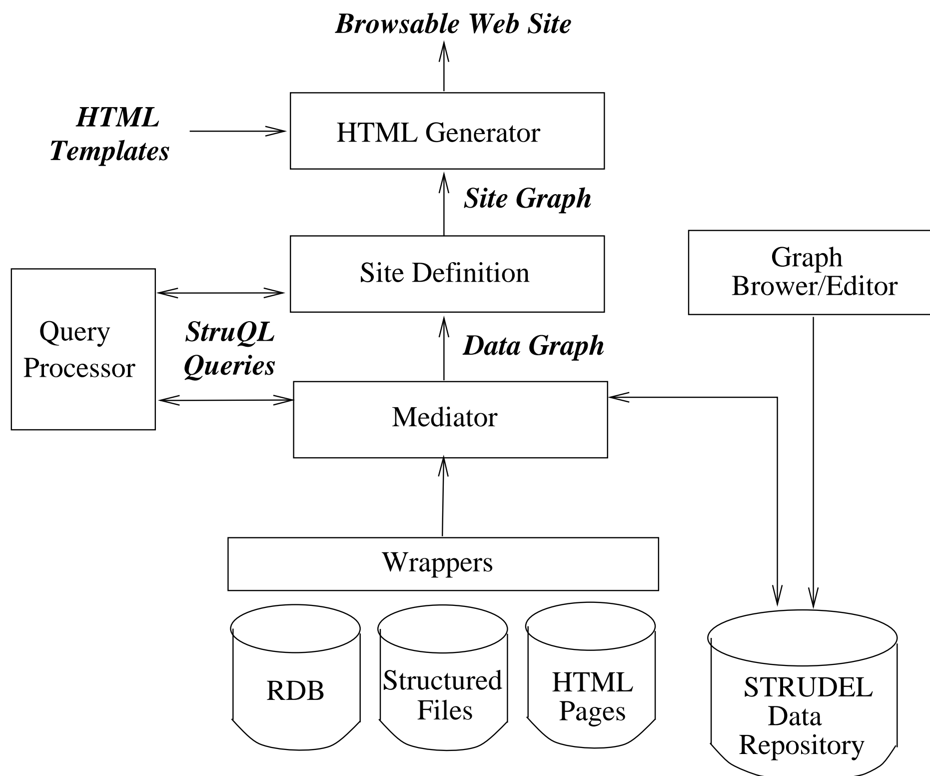

Figure 1: STRUDEL Architecture

2.1 System Architecture

Data model: In every level of the STRUDEL system, we use a data model based on labeled directed graphs; our model is similar to OEM, which was developed in the TSIMMIS project [6]. The labeled graph model has been proposed for managing semistructured data, which often has few type constraints, a rapidly evolving schema, or missing schema. It has been shown that the semistructured data model also facilitates data integration.

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, ASCII, 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 is similar to OEM's data definition language [6].

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, our solution was to 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. In Section 4, we discuss the lessons we learned from using our data repository.

Mediator: One of STRUDEL's key benefits is the ability to build Web sites that serve data from multiple sources. To support this functionality, STRUDEL includes a mediator that provides a uniform view of all the underlying data, irrespective of where it is stored. When designing the mediator, we considered two main issues that arise in data-integration applications. The first is whether to perform data integration by warehousing data from external sources or to access the external sources on demand at query time (see [13] for a comparison of the two approaches). The second issue is how to specify the relationship between the attributes and collections in the mediated schema and those in the data sources (see [17] for a discussion of the possible approaches).

In our prototype implementation, we implemented the warehousing approach. The result of the data integration is stored in STRUDEL's data repository. This choice simplified our implementation and sufficed for our applications; STRUDEL's architecture, however, does not preclude a virtual approach.

Recent research addresses the problem of specifying the relationship between the mediated view of the data and the external data sources. Two general approaches are advocated: *Global as view* (GAV) [6, 2, 12, 10, 16] and *Local as view* (LAV) [15, 14, 7, 8]. STRUDEL currently uses the GAV approach. We justify our choice in Section 4.

Query processor: STRUDEL provides a novel language, STRUQL, for querying and restructuring semistructured data. Since data graphs and site graphs are represented in the same data model, i.e., labeled graphs, STRUQL queries can be applied to any graph, whether it was 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 [11]. 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 graphical presentation of 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 in the graph 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. Section 2.4 describes the HTML template language in detail.

Data repository browser/editor: The repository browser and editor enables a user to create, update, and view graphs and can be used for both the data graph and site graphs. A graph is viewed one object at a time and is browsed by following links that are the object's attributes. The user can also specify simple selections on a graph's objects and view the result. The editor also provides a user-friendly interface for creating new graphs. This is important in our application, because often the data to appear in the Web site is not available online and must be entered manually.

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), at both levels. A query in StruQL's core fragment has the form:

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

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

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

HomePages is a collection, "*Paper*" is an edge label, and *isPostScript* is a predicate testing whether node q is a PostScript file. The distinction between collection names and external predicates is done at a semantic, not syntactic, level. Strudel maintains lists of collection names and of available external predicates. 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 edges, e.g., $isPostScript(q)$. A condition of the form $x \rightarrow R \rightarrow y$ means that there exists a path from node x to node y that matches the regular path expression R. 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:[1]

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

New is a Skolem function that creates new object 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. For each node q and q', it constructs new nodes $New(q)$ and $New(q')$. By definition, a Skolem function when applied to the same inputs returns the same node oid, so 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 $Home$ link that points to the new root. Finally, it creates an output collection $TextOnlyRoot$ that contains the new graph's root.

StruQL's semantics can be described in two stages. The *query stage* depends only on the **where** clause and produces all possible bindings of node and edge variables to values that satisfy all conditions in the clause; its result is a relation with one attribute for each variable. The *construction* stage constructs a new graph from this relation, based on the **create**, **link**, **collect** clauses. We adopt an *active-domain* semantics for StruQL. The meaning of the **where**-clause is the set of assignments from variables to oid and label values 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 mentioned in **link** or **collect** is mentioned in **create** or is a node in the data graph and that edges are added from new nodes to new or existing nodes; existing nodes are immutable and cannot have edges added to them. Strategies for efficient evaluation and optimization of StruQL queries are described elsewhere [9].

StruQL's semantics provides an unexpected benefit. Because StruQL's query and construction stages are separate, a simple analysis of the query can infer the *graph schema*[4] of the result graph. Graph schemas are useful for viewing a skeleton of the result graph, which makes them invaluable tools during the iterative definition of a Web site's structure. Graph schemas also allow the user to visually verify a site's integrity constraints (e.g., connectedness, reachability of certain nodes from others.)

2.3 Example Web Site

The following example shows how the site graph for one author's homepage is generated by Strudel.[2] The main source of data for his homepage is his Bibtex bibliography file. The homepage site has four types of pages: the root page containing general information, an "abstracts" page containing all paper abstracts, a "year" page containing summaries of papers published in a particular year, and a "category" page containing summaries of papers in a particular category. 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. In Section 2.4, we describe generation of the site's browsable HTML pages.

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

[2] We encourage the reader to visit the generated site at `http://www.research.att.com/~levy`.

```
collection Publications {
  abs-file text
  ps-file  ps
}
object pub1 in Publications {
  title "CARIN: A Representation ..."
  author "Alon Y. Levy"
  author "Marie-Christine Rousset"
  year   1996
  booktitle "Proc. of European ..."
  pub-type  "inproceedings"
  abs-file  "abstracts/ecai96"
  ps-file   "proceedings/ecai96.ps"
  category  "Description Logics"
  category  "Datalog Optimization"
}
```

```
object pub2 in Publications {
  title  "Data model and query ..."
  author "Alon Y. Levy"
  author "Divesh Srivastava"
  author "Thomas Kirk"
  year   1995
  month  "September"
  journal  "Journal of Intelligent..."
  pub-type "article"
  abs-file "abstracts/jiis95"
  ps-file  "proceedings/jiis95.ps.Z"
  volume   "5 (2)"
  category "Data Integration"
  category "Relevance Reasoning"
  category "Materialized Views"
}
```

Figure 2: Fragment of data graph for example homepage site

Figure 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, *pub2* has a *month* attribute but *pub1* does not; *pub1* has a *booktitle* attribute, whereas *pub1* has a *journal* attribute. By default, STRUDEL infers the types of attribute values from their format, e.g., integer, float, string. The *collection* directive specifies the default types of attribute values that would otherwise be interpreted as strings, e.g., *abs-file* is a text file and *ps-file* 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 Figure 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 member 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 *PagePresentation* 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.

Figure 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.

The choice to realize internal objects as pages or as page components is delayed until HTML

```
1    INPUT BIBTEX
2    // Create root page and 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
8    CREATE PaperPresentation(x), AbstractPage(x)
9
10   LINK   AbstractPage(x) -> l -> v,
11          PaperPresentation(x) ->  l -> v,
12          PaperPresentation(x) -> "Abstract" -> AbstractPage(x),
13          AbstractsPage() -> "Abstract" -> AbstractPage(x)
14     // Create a page  for every year
15     { WHERE  l = "year"
16       CREATE YearPage(v)
17       LINK   YearPage(v) -> "Paper" -> PaperPresentation(x),
18              YearPage(v) -> "Year" ->  v
19
20              // Link root page to each year page
21              RootPage() -> "YearPage" -> YearPage(v)
22     }
23     // Create a page  for every category
24     { WHERE  l = "category"
25       CREATE CategoryPage(v)
26       LINK   CategoryPage(v)  -> "Paper" -> PaperPresentation(x),
27              CategoryPage(v)  -> "Name" ->  v,
28
29              // Link root page to each category page
30              RootPage()  -> "CategoryPage"  -> CategoryPage(v)
31     }
32   OUTPUT HomePage
```

Figure 3: Site definition query for example homepage site

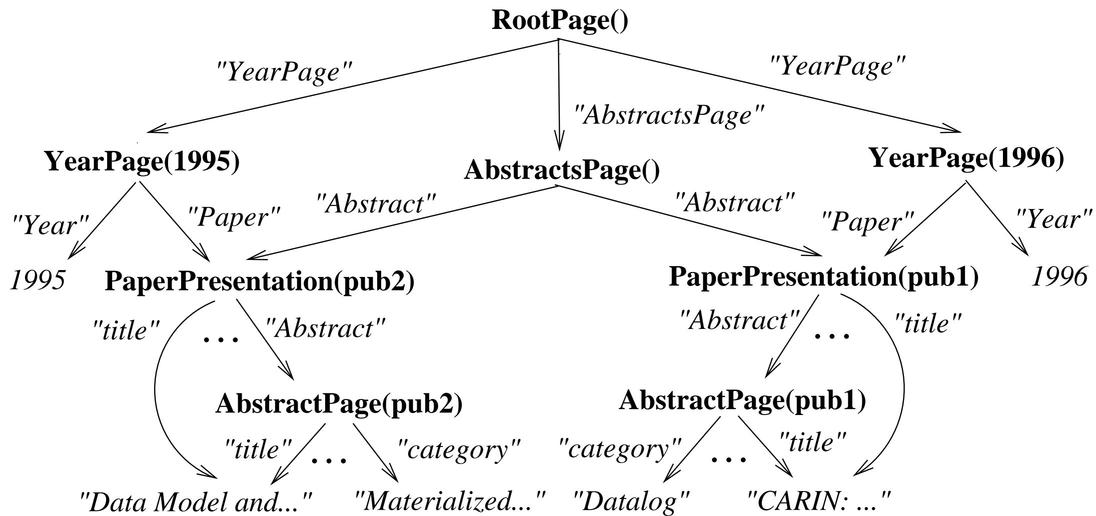

Figure 4: Fragment of site graph for example homepage site

generation; our choice of Skolem function names (e.g., *AbstractsPage*) hints that some objects will be realized as pages, but this is not a requirement. For example, when referenced from the *PaperPresentation* template, an *AbstractPage* object is realized as a separate HTML page, but when referenced from the *AbstractsPage* template, an *AbstractPage* object is embedded in the generated HTML page.

2.4 HTML-Template Language

One premise of STRUDEL's design is that designing and *changing* the graphical presentation of a site should be completely separable from the management of the site's content and structure. Although HTML generation was not our central concern, we wanted STRUDEL users to be able to easily produce visually consistent and attractive sites. The result is a template language that extends plain HTML files with three simple expressions; this design evolved from our experience of representing sites in a semistructured data model.

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. The template file is either (1) an object-specific file whose name contains the object's unique identifier, (2) the value of the object's "HTML-template" attribute, or (3) the HTML 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. Figure 5 contains the HTML templates for the example site. The *RootPage*, *AbstractsPage*, *YearPage*, *CategoryPage* and *AbstractPage* are realized as pages.

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 simple extensions to plain HTML: a format expression, a conditional expression, and an enumeration expression, each of which produces plain HTML text.

RootPage template:

```
<html>
<!-- Raw HTML text omitted -->
<h2>
<$AbstractsPage LINK="All Paper Abstracts">
</h2>
<h2> Publications by Topic </h2>
<$CategoryPage LINK=CategoryPage.Name UL
  ORDER=ascend KEY=Name>

<h2> Publications by Year </h2>
<$YearPage LINK=YearPage.Year UL
  ORDER=ascend KEY=Year>
<!-- More raw HTML text omitted -->
</html>
```

AbstractsPage template:

```
<html>
<H1>Paper Abstracts</H1>
<$Abstract EMBED UL>.
</html>
```

YearPage template:

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

CategoryPage template:

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

PaperPresentation template:

```
<$ps-file LINK=title>.
By <$author ENUM DELIM=", ">.
<i>
<SIF booktitle> <$booktitle>
<SELSE journal> <$journal>
</SIF>, <$year>.
</i>
<$Abstract LINK="Abstract">
```

AbstractPage template:

```
<$ps-file LINK=title>.
By <$author ENUM DELIM=", ">.
<i>
<SIF booktitle> <$booktitle>
<SELSE journal> <$journal>
</SIF>, <$year>.
</i> <BR><BR>
<$abs-file EMBED><BR><BR>
```

Figure 5: HTML Templates for example homepage site

```
Template ⇒ {(Extension | Plain HTML Text)}

Extension ⇒ <SFMT AttrExpr [Format] [Enumerate]>
          | <$ AttrExpr [Format] [Enumerate]>
          | <SIF CondExpr > Template [<SELSE> Template] </SFI>
          | <SFOR ID IN AttrExpr [Order] > Template </SFOR>

AttrExpr ⇒ (@ ID [{. ID}] | ID [{. ID}])

Format ⇒ (EMBED | [LINK [= Tag]]) [ARGS= Tag]

Tag ⇒ {(STRING | AttrExpr)}

Order ⇒ ORDER=(ascend | descend) KEY=ID

Enumerate ⇒ (OL | UL | ENUM) [DELIM=STRING] [Order]
```

Figure 6: EBNF Grammar for HTML-Template Language

Figure 6 contains the language's grammar.

The format expressions (<`SFMT`...> and < $ ... >) map 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., `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 the modularity of the queries.

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, <`$Year`> is replaced by the object's `Year` attribute, which is an integer. Some atomic values, such as PostScript and image files, should not be realized as strings. For these values, the HTML generator produces an appropriate link to the value. For example, in the *PaperPresentation* template, <`$ps-file LINK=title`> is replaced by a link to the object's `ps-file` 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 object is not designated as a page or with a link to the object's HTML file if it is designated as a page.

The user can override the HTML generator's default actions. The `LINK` directive makes linking explicit; the optional `LINK` value is used as the tag text for the link. By default, the attribute's string value is used as the tag text. The `EMBED` directive embeds the attribute's HTML value. For example, in the *AbstractsPage* template, the HTML values for all `Abstract` objects are embedded in the page; by default, these objects are realized as pages, because they are members of the *AbstractPage* collection.

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 the HTML template. The conditional expression `SIF` evaluates a condition and if it is true, evaluates the first template expression, otherwise it evaluates the optional second template 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`, `ps-file`, `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., the `RootPage` object 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 *ObjVar* to each value, and evaluates the nested template expression for each binding. *ObjVar* 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>
   <SFMT @a EMBED>,
</SFOR>
```

Enumerating all values of an object's attribute is common, so we abbreviate common idioms. For example the expression `<$author ENUM DELIM=", ">` in the *PaperPresentation* template is equivalent to the expression above. Attributes are often emitted in ordered and unordered lists. For example, `<$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 semistructured graph 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 `<$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 list.

3 The Outstanding Questions

After implementing the first prototype of STRUDEL, we wanted to validate STRUDEL's benefits, and more generally, evaluate the impact of STRUDEL's core ideas. Our goal was to answer the following questions:

STRUDEL's **methodology:** The main premise of STRUDEL's design is that the three tasks of Web-site creation (management of the underlying data, management of the site structure, and graphical presentation of the site) *can* and *should* be separated. The primary question is whether this premise holds in practice. Specifically,

- Is there always a clear separation between these tasks? If not, in which cases are they so interrelated that their separation becomes counter productive? When does this separation simplify Web-site creation and maintenance?

- Under what circumstances is STRUDEL most useful, i.e., for what kinds of Web sites is STRUDEL providing important advantages? How useful is the ability to explicitly and declaratively manage a Web site's structure?

Advantages of the semistructured data model: STRUDEL is based on a semistructured data model, i.e., a labeled, directed graph. This choice was appealing for two reasons. First, Web sites are naturally viewed as graphs of nodes (i.e., pages). Second, Web-site creation often requires integrating data from multiple sources, and a semistructured data model is well suited to data integration. We addressed the following issues:

- Semistructured data is promoted as especially appropriate for applications in which the structure of the data is irregular or partial, the data has few type constraints, and the data's schema evolve rapidly. What characteristics of semistructured data were most important in STRUDEL? Conversely, why could we not effectively implement STRUDEL in a traditional data model?

- STRUQL provides features associated with query languages for semistructured data, e.g., regular path expressions and restructuring capabilities. Are these features necessary or useful for STRUQL's primary task, which is site definition? What features are missing from STRUQL that might simplify site definition? In general, w.r.t. this task, how well does STRUQL compare with other languages for querying semistructured data?

STRUDEL **as an application of data integration:** Web-site creation often requires integrating data from multiple sources, some of which may not be database systems. What are the necessary characteristics of a data-integration system when used in a Web-site management application? Has our solution to data integration in STRUDEL more clearly identified the design tradeoffs addressed in research on data integration?

4 Experiences with STRUDEL

We have had both practical and exploratory experiences with STRUDEL. In our practical experience, we have 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 our own demonstration purposes. In our exploratory experience, we have described our methodology and demonstrated our prototype to several potential STRUDEL users. We describe this experience, addressing the above questions wherever relevant.

4.1 Practical experience

Our largest examples to date are the internal and external Web sites of AT&T Labs–Research. We have built versions of these sites that are identical to those built by our Web administrators. The official external site is at `http://www.research.att.com` and is generated using a large set of CGI-BIN scripts; the STRUDEL-generated version is at `http://www.research.att.com/~mff/external`. This site is typical of an organization's site: it includes home pages of individual members, pages on projects, demos, and research areas, technical publications, and student programs. 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 perform some data integration, such as linking associated objects in *Person*, *Organization*, and *Project* collections. The wrappers for plain HTML pages are written by hand.

The internal site generated by STRUDEL contains the home pages of approximately 400 users and pages for organizations and projects. The interanl site is defined by a 115-line query and 17 HTML templates (380 lines). The power of STRUDEL 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 (e.g., `http://www.research.att.com/{~levy,~mff}`) are examples of small sites generated by STRUDEL. The main data sources for sites are our bibliographies. We have a simple wrapper that maps BIBTEX files into data graphs; other information is stored in files in our data definition language. The example in Section 2.3 describes creating the example homepage site. The `mff` example shows how a single page can include information derived from two queries: the top part of the page is generated by a query defined by the Web administrator and is identical for all users; the lower part of the page is generated by a user-defined query. The `levy` pages are defined by a 24-line STRUQL query and six HTML templates (168 lines); the `mff` pages are defined by a 20-line query and two HTML templates (54 lines).

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 distinguishing feature is that there are two views, one English and one French, of the site. The two sites are cross-linked 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 site 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, which makes the site very suitable to generation by STRUDEL. Because we did not have access to CNN's databases of articles, we mapped their HTML pages into a data graph containing around 300 articles. Our version of the CNN site is defined by a 44-line query and nine templates. To demonstrate STRUDEL's ability to easily generate multiple sites from one database, we also generated a "sports only" site that has the same structure as the general site but contains only articles on sports subjects. The query to generate the sports-only site is derived from the original query and only differs in two extra predicates in the **where** clause. The same HTML templates are used in both sites. `http://www.research.att.com/~levy/strudel-demo.html` contains the STRUDEL-generated versions of the CNN site and the CNN Sports site.

4.2 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 for others, and a company that creates Web-based interfaces and data-integration technologies for business applications. Not suprisingly, 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 significant 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 not only allow the user to organize his news as he wanted, but would allow CNN to generate pages that contain advertising content targeted for that user. STRUDEL's separation of site management and graphical 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.

The 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 automatically propagated back into the HTML templates, site-definition query, or underlying data. Not surprisingly, several customers noted that a graphical interface for specifying STRUQL queries in the spirit of Query By Example [19] would be necessary.

In summary, our experience validated the important ideas in STRUDEL, showed us that even a prototype implementation of STRUDEL is useful, and identified the problems that would have to be solved in a production quality implementation.

5 The Lessons Learned

We describe the lessons we learned from our experience using STRUDEL and identify the major questions that must be addressed in an industrial quality implementation.

5.1 The STRUDEL Methodology

Separating the three tasks of Web-site creation proved very important. We examine several aspects of this separation.

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, required integration of five data sources.

Isolating the management of a site's structure was also important. In our experience, we found that managing site structure is increasingly important to Web-site builders. For example, CNN has a large programming group building a specialized tool for managing their 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 graphical 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 graphical presentation varies. 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 graphical 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 do not emit these attributes than it is to create a new site graph that explicitly excludes those attributes. Consider

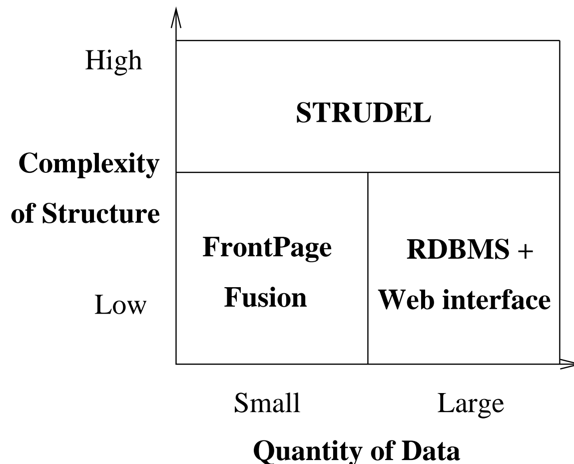

Figure 7: Suitability of Web-site management tools

the order of articles or the placement of images in a page at the CNN site. Such information could be encoded in the graphical 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 better handle the separation between structure and presentation, we created the HTML-template language described in Section 2.

We want to characterize the sites for which STRUDEL is most useful. To do so, we classify Web sites according to two criteria: the amount of data they contain and the complexity of their structure (see Figure 7). Measuring the amount of data in a site is straightforward, but measuring the complexity of a site is more subtle. One possible measure is the ratio the number of pages in the site to the size of the site-definition query (e.g., the number of link clauses). A large ratio indicates that the structure is replicated. 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 provides maximal benefit for sites with 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 quite 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.

Figure 7 categorizes the suitability of different Web-creation tools for various kinds of sites. When a site contains little structure and little data (lower left), WSYWIG tools such as Microsoft FrontPage or NetObjects Fusion are appropriate choices. 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. However, when the structure is complex and there is a lot of data, STRUDEL would be most appropriate.[3] Finally, we emphasize that our experience with Web-site design companies indicated that sites with large amounts of data and complex structures are increasingly common.

[3] We did not see sites that have little data and a lot of structure. If such sites exists, and as explained above, the structure is not as big as the data, then STRUDEL would be appropriate.

5.2 Semistructured Data

We chose a semistructured data model for STRUDEL, because it is natural to model Web sites as graphs and because we needed to integrate data from multiple, non-traditional sources. Our experience indicates that the semistructured model was the right choice, and that STRUQL is suitable for the application. The most important features of semistructured data are the abilities to easily evolve the schema and to manage irregular data. We explain these and other points below.

Schema evolution: 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. Recall that we build two graphs: the data graph and the site graph. Creating both 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. For example, 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. A site graph's schema evolves as the site is defined. 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 the schema for site and data graphs, we conclude that traditional database systems would not be suitable data repositories.

Irregular structure: In the semistructured data model, we associate sets of objects with *collections*. A collection is like a class, except that objects need not have the same representation, i.e., the same attributes or the same type for each attribute. 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.

The StruQL query language: We were pleasantly surprised by how well StruQL was suited for the application. StruQL's most important feature is the separation of the selection stage (where clause) and the construction stage (create, link, and collect clauses). This separation is natural, because it is conceptually clearer to separate extraction of data from external sources and site construction. Another advantage is that the selection stage is independently extensible; for example, we could extend it to include grouping and aggregation. This separation also simplifies query optimization, because all selection clauses can be evaluated by the optimizer at once. It should be noted that other languages (e.g., UnQL [5]) do not separate selection and restructuring. The Lorel query language cannot express StruQL's graph restructurings, and in particular, not those required in our application.

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.

In our experience, we found that StruQL queries tend to be long. To simplify writing queries, we made several changes to the language. First, we introduced nested queries as described in Section 2. Second, 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 a site. Finally, we built a tool to view a query's graph 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 our example query in Figure 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. We also found that to better query schema, StruQL should support collection variables; collection variables are needed for simple operations such as copying parts of an input graph into the result graph.

Storage for semistructured data: In Section 2, we briefly described Strudel's data repository. One drawback to our data repository is that it cannot adapt a graph's storage to a particular application. All graphs are stored in the same representation and provide the same set of indexes. Obviously, different applications may require different storage schemes. To support this, we need a language for specifying storage representations (e.g., the set of indexes, forward vs. inverse links) that accommodate frequent data-access patterns. Designing such a language is non-trivial, because we do not have an explicit schema. The data repository should also support flexible indexing schemes.

Other features of semistructured data: Abiteboul [1] identifies the salient characteristics of semistructured data. Some of these characteristics are mentioned above. For completeness, we mention the other characteristics listed in [1] that were *not* important in our application and explain why this was the case.

Abiteboul states that the schema for semistructured data is often implicit in the data, may be very large with respect to the data, or simply be ignored. We did not encounter these characteristics in our application. Except for the irregularities mentioned above and that sophisticated format translations may be performed by the wrappers, our data sources were fairly structured. For example, we did not attempt to *extract* data from existing HTML pages. We expect that most Strudel users will have access to the data underlying their Web sites, not only to HTML renderings of the data. Although Strudel's output is a set of Web pages, its input is relatively structured data, and knowledge of this structure is needed to construct the Web site.

Because the data sources were structured, our site-definition queries rarely used the closure operator or wildcard matching supported by 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 applications. We expect, however, that regular path expressions will be useful in other applications of STRUQL. For example, regular path expressions are necessary to express integrity constraints on a Web site, e.g., to state that all pages must be reachable from the site's root or that every department member can be reached from a department page. Regular path expressions also may be useful for querying a site constructed by STRUDEL.

Extending the data model: 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 in our data model. A feature of the data model whose need we did not anticipate was ordered lists. For example, objects in the *Publications* collection have an associated *list* of authors. Maintaining the 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.

We conclude that the semistructured data model offers several benefits that were necessary in STRUDEL. We consider the semistructured data model flexible enough to support data that lacks structure, but expressive enough to utilize structure when it exists. STRUDEL is an appropriate and important application of semistructured data.

5.3 Data Integration

One of STRUDEL's benefits is that it allows the Web-site builder to display information derived from multiple data sources. Often, the sources are only partially structured (e.g., structured files). We address two issues related to data integration in STRUDEL.

Warehousing vs. virtual. Warehousing and virtual are two common approaches to data integration (see [13] for a comparison). In the warehousing approach, data from multiple sources is loaded into a warehouse, and all queries are applied to the warehoused data; this requires that the warehouse be updated when data changes. In the virtual approach, the data remains in the sources, and queries to the mediator are decomposed at runtime into queries on the sources. STRUDEL currently implements the warehousing approach. The reason for this choice is that it simplified our implementation and sufficed for our applications, which tended to deal with small databases. However, there are cases when warehousing is infeasible due to the sources' large size or high frequency of update. Since we anticipated the need for a virtual approach, STRUDEL's architecture can accommodate either approach.

We highlight the important research issues that need to be addressed in production implementations of both approaches. A simple implementation of warehousing reloads the warehouse whenever the underlying data changes. For warehousing to be feasible in applications in which the data changes frequently, we need to solve the problem of *incremental view updates for semistructured data,* which is currently an open problem.

Implementing the virtual approach requires that we solve the problem of *translating a query on the 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 in StruQL). introduce difficulties.

Specification of mediated schema. The mediated schema in a data integration system is the set of collection and attribute names that are mentioned in the users' queries. To evaluate a query, the data integration system must translate the query on the mediated schema into a query on the data sources, that have their own local schemas. Recent research addresses the problem of specifying the relationship between the relations in the mediated schema and those in the external sources. Two general approaches are advocated: *Global as view* (GAV) [6, 2, 12, 10, 16] and *Local as view* (LAV) [15, 14, 7, 8]. These approaches are discussed for relational mediated schemas and source schemas.

In the GAV approach, the relationship between the two types of relations is specified by a set of queries. For each relation R in the mediated schema, we write a query over the source relations specifying how to obtain R's tuples from the sources. The LAV approach is the inverse. For every information source S, we write a query over the relations in the mediated schema that describes how R's tuples can be found in S. Ullman [17] compares the two approaches and identifies the advantages of both. GAV's main advantage is that it provides finer control over how to combine the data from the sources. In contrast, the LAV approach simplifies adding and deleting sources and accommodates sources with overlapping data.

We found the GAV approach was suitable for our application, because the number of data sources we integrated was usually small, and the set of sources 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 relations in the sources frequently. We opted for the GAV approach, because it was immediately extensible to our query language.

5.4 Research Issues

Our experience raised several interesting research issues, which we describe here.

- **Graphical interface to the query language.** Not surprisingly, many people who reviewed Strudel asked whether we 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 [19]. 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^{+} [18].

- **Computing incremental updates.** Another research issue is how to update the Web site when the underlying data changes. Obviously, recomputing the entire site graph after each change is infeasible, therefore we need to develop algorithms for incremental view update of semistructured data. Our instance of this problem is more complex than previous work on incremental update, because StruQL can also query the schema, restructure the data, and query regular path expressions.

- **Incremental generation of the Web site.** In Strudel's current implementation, we precompute a Web site by materializing the site graph and all the HTML pages. In some Web sites, however, pages need to be generated dynamically, as the user browses the site. Currently, this functionality is support by CGI scripts. Supporting dynamic generation with Strudel requires evaluating the relevant parts of the site-definition query as each page is

browsed. The key problem is how to *isolate* the relevant subquery for a single object in the site graph; this subquery should be evaluated dynamically when the corresponding page is requested.

- **Storage for semistructured data.** Traditional database systems rely heavily on schema information to organize the data on disk. A major question is how to develop analogous techniques for semistructured data where schema information is missing or changes frequently. In a sense, this is a more complex version of the problem of dynamic reclustering of objects in object-oriented databases. Traditional systems also use query patterns to decide which indexes to build. However, in STRUDEL, identifying query patterns is complicated by the novel features of STRUQL(e.g., querying schema).

- **Integration with the programming environment.** 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. In particular, developing the appropriate API to STRUDEL may be the best way to incorporate it into tools that Web-site builders currently use.

6 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 management technologies. We also recognized that separating the management of a site's data, the management of its structure, and the graphical 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 open 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 plan to address these problems in future work on STRUDEL.

References

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

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

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

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

[5] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of SIGMOD–96*, pages 505–516, 1996.

[6] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogenous information sources. In proceedings of IPSJ, Tokyo, Japan, October 1994.

[7] Oliver M. Duschka and Michael 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.

[8] Oliver M. Duschka and Michael R. Genesereth. Query planning in infomaster. In *Proceedings of the ACM Symposium on Applied Computing, San Jose, CA*, 1997.

[9] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. System demonstration - strudel: A web-site management system. In *ACM SIGMOD Conference on Management of Data*, 1997.

[10] 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.

[11] Daniela Florescu, Alon Levy, and Dan Suciu. A query optimization algorithm for semistructured data. Submitted for publication, 1997.

[12] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd VLDB Conference, Athens, Greece*, 1997.

[13] Richard 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.

[14] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, 1996.

[15] 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.

[16] 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.

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

[18] 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.

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