Proceedings of DSL'99: The 2nd Conference on Domain-Specific Languages

Austin, Texas, USA, October 3–6, 1999

DECLARATIVE SPECIFICATION OF DATA-INTENSIVE WEB SITES

Mary Fernández, Dan Suciu, and Igor Tatarinov

Declarative Specification of Data-intensive Web sites

Mary Fernández[*]
AT&T Labs - Research
mff@research.att.com

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

Igor Tatarinov
North Dakota State University
tatarino@prairie.NoDak.edu

Abstract

Integrated information systems are often realized as *data-intensive Web sites*, which integrate data from multiple data sources. We present a system, called STRUDEL, for specifying and generating data-intensive Web sites. STRUDEL separates the tasks of accessing and integrating a site's data sources, building its structure, and generating its HTML representation. STRUDEL's declarative query language, called StruQL, supports the first two tasks. Unlike ad-hoc database queries, a StruQL query is a software artifact that must be extensible and reusable. To support more modular and reusable site-definition queries, we extend StruQL with functions and describe how the new language, FunStruQL, better supports common site-engineering tasks, such as choosing a strategy for generating the site's pages dynamically and/or statically. To substantiate STRUDEL's benefits, we describe the re-engineering of a production Web site using FunStruQL and show that the new site is smaller, more reusable, and unlike the original site, can be analyzed and optimized.

1 Introduction

In large corporations, high-speed intranets and Web browsers have increased the demand for integrated information systems. Before intranets, access to geographically dispersed information systems was usually limited to those people who administered the systems locally. In this environment, *data integration*, the task of integrating information from multiple data sources, was difficult, if not impossible. An AT&T customer, for example, may have multiple accounts, e.g., long distance and wireless, stored in separate account-management systems. An integrated "view" of customers' accounts is vital to many business processes, such as targeting new services to appropriate customers. Because of their value to diverse groups, integrated information systems must be easily accessible and therefore, are usually realized as Web sites. These systems, which we call *data-intensive* Web sites, integrate information from multiple data sources, often have complex structure, and present increasingly detailed views of data, from a summary perspective at a top-level page to a detailed perspective at a lower-level page.

In previous work [9], we argued that building data-intensive Web sites is a data-management problem, whose solution consists of three main programming tasks: *accessing and integrating* the data available in the site, *building* the site's structure, i.e., specifying the data in each page and the links between pages, and *generating* the HTML representation of pages. To better support these tasks, we developed the STRUDEL system, which applies concepts from database-management systems to Web-site creation and management. STRUDEL's key idea is separating the management of a Web site's data, the specifications of its structure, and the HTML representation of its pages. STRUDEL provides a declarative *query language*, StruQL, for specifying the content and the structure of a Web site, and a simple *template language*, for specifying a site's HTML representation. STRUDEL's query interpretor automatically derives the site from a StruQL query. STRUDEL has many benefits: explicit separation of the three programming tasks allows multiple versions of a site to be derived from one specification [9], and StruQL's semantics supports verification of integrity constraints on a site's structure [10].

In this paper, we argue that building data-intensive Web sites is also an important *software-engineering* problem, and that a site's implementation, like other valuable software systems, should be extensible, reusable, and optimizable. For example, it should be easy for a site developer to integrate new data sources into a site and to derive a new site from an existing one. It should also be possible for the site developer to optimize overall site performance, for example, by using page-access patterns culled from a Web server to drive static and/or dynamic generation of the site's pages, or by identifying au-

[*]Contact author: Mary Fernandez, AT&T Labs - Research, 180 Park Ave., Room E243 Florham Park, NJ 07932-0971, 973-360-8679, (FAX) 973-360-8077

tomatically pages that contain data from the same sources and by caching shared data when it is expensive to compute. Site reuse is greatly simplified if the implementation clearly separates the definition of the site's content, structure, and presentation. Both the site-generation and data-caching problems are examples of site optimizations and are orthogonal to the site's definition. For example, choosing a site-generation strategy is analogous to optimizing a program given an execution profile.

In current practice, most data-intensive Web sites are implemented by loosely related programs written in imperative scripting languages, such as Perl. Scripting languages are well-suited for "gluing" together other software components [15], which makes them popular for constructing Web sites. The scripts for many site implementations, however, interleave the code for data access and integration, page construction, and HTML generation. Even when these tasks are separated, it is difficult to infer automatically the semantics of the script code. Interleaving of these tasks limits reuse of any one component and prevents analysis of the site implementation as a cohesive unit. Finally, the site-generation and data-caching strategies are usually encoded explicitly in the implementation, making it difficult to experiment with alternative policies.

In this paper, we show that StruQL is more effective than general-purpose scripting languages for implementing data-intensive Web sites. StruQL is an example of a *declarative query language*, and although it is not Turing complete, it has been used to implement several Web sites[1]. Unlike higher-level programming languages, declarative query languages (e.g., SQL, OQL) usually express short, adhoc queries. They lack features, such as functions and modules, that support development of large software systems, which must be modular, extensible, and reusable. STRUDEL's application requires both the declarativeness of query languages and the functional constructs of higher-level programming languages. The main contribution of this paper is the integration of these features in one language. In particular, we extend StruQL with *functions* to improve the modularity and re-usability of site definitions and with *forms* to support dynamically bound inputs. We describe how the new language, called FunStruQL, supports flexible site-generation strategies and how forms can be specified declaratively.

[1]We encourage the reader to visit the STRUDEL-generated sites at `http://www.research.att.com/~mff,~suciu` and `http://www.research.att.com/conf/sigmod99`.

We support these claims through a case study of an internal AT&T Web site that is used in a production setting. We compare the site's original implementation with its complete re-implementation in STRUDEL and show that the new implementation is much smaller, more reusable, and unlike the original site, can be analyzed and optimized.

1.1 Case Study

To motivate STRUDEL's design, we first describe an internal AT&T Web site, called "hightoll notifier" (HTN), which identifies business-customer accounts that appear to be high risk, i.e., ones whose bills may go unpaid. Statistically, customers that have a significant increase in their telephone usage over a short period of time are more likely to not pay their bills than those customers that have constant daily usage. Other high-risk indicators include the customer's credit record and their ability to pay previous bills on time.

The data in the HTN site must be current to within one day or even a few hours, so that account representatives can identify and contact high-risk accounts before the account further increases its usage or goes into arrears at billing time. Before the HTN site existed, account representatives might have waited several weeks before they had sufficient information to identify high-risk accounts. The HTN site is a tremendous success, because it provides in real time an integrated view of high-risk accounts.

HTN is a good example of a data-intensive Web site: it integrates data from multiple sources and allows the site user to "drill down" from the high-level, summary perspective to the low-level source data. HTN computes usage statistics on approximately 250 million phone calls daily and integrates information from several sources: phone-call records, long-term account information, and credit records. Of the 1.7 million business accounts tracked, approximately 6000 are identified as potential risks. The site has five levels: each subsequent level provides a more detailed view of the high-risk accounts. The *root page* allows an account representative to select the types of high-risk accounts to track, e.g., a particular market segment. The *hot list* page lists the set of accounts in the chosen segment and orders them by a risk metric. The hot-list page points to *account pages*, which displays a summary of an account's usage in textual and graphical form. A *report page* is accessible from several pages in the site and presents the account's risk metrics and allows the account rep to view and record interactions with the customer. The most detailed page presents the original phone-call records from which the usage summary is computed.

The first HTN site was implemented using scripting languages, e.g., Korn shell and Perl, and common Unix command-line tools, e.g., awk, sed, and grep. The scripts process user inputs, invoke Unix tools to handle simple data-management tasks, and format and emit HTML pages. Several C programs implement rudimentary database operations. Although some scripts differentiated the three site-creation tasks, most scripts interleaved them. The result is a loosely related set of scripts that implement the required functionality, but that have the characteristics of a poorly implemented software system: the code is hard to understand and extend, because the program's tasks are undifferentiated. These problems complicated extension and prevented reuse of HTN's first implementation.

Given these limitations, the site was re-engineered using Strudel [9] and the Daytona relational database management system [13]. The short-term goal was for the new implementation to simplify maintenance and extension of the site. The long-term goal was to show that declarative specification supports site reuse and flexible site-generation strategies. Overall, the Strudel implementation is 1.6 times smaller than the original implementation, but if we compare only the code that defines the site's content and structure, i.e., the code that a developer must understand to reuse or extend the site, it is more than 4 times smaller. Section 6 presents an evaluation of this re-engineering effort.

2 Strudel **Overview**

We first describe Strudel's architecture, depicted in Figure 1, before focusing on its query language. Rectangles depict processes and emboldened terms specify the inputs and outputs of the processes.

Strudel supports two common types of Web-site data: *semistructured data* and *tuple-stream data* (bottom of Fig. 1). Semistructured data is characterized as having few type constraints, irregular structure, and rapidly evolving or loosely defined schema [1]. Web sites and XML data [7] are good examples of semistructured data. For example, XML elements can have missing attributes or attributes whose value is not strictly typed (e.g., a `name` attribute may have an atomic value in one element

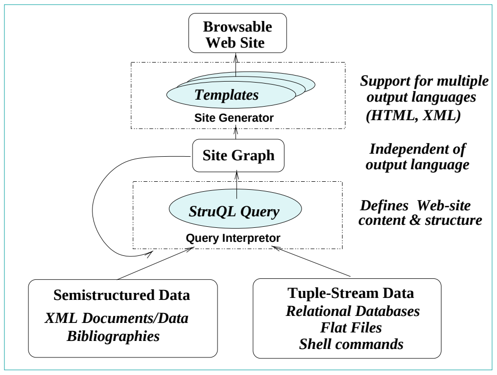

Figure 1: Strudel Architecture

and a complex value, `(lastname, firstname)`, in another element.).

As in related systems [8], Strudel represents semistructured data as a *collection* of objects, in which each object is either *complex* or *atomic*. A complex object is a set of *(attribute, object)* pairs, and an atomic object is an atomic value (e.g., `int`, `string`, `video`). Hence, data is a *graph*, with edges labeled by attributes and leaves labeled with atomic values. Internal nodes have unique object identifiers, called *OIDs*, and data can be exchanged in a text representation. For example, Fig. 2 contains a fragment of an address database in XML. The complex objects `addresses` and `entry` have object identifiers (the `id` attribute). In semistructured data, the names, types, and cardinality of attributes may vary. For example, the first `entry` element has two `street` attributes, but the second has only one; the second has a `postalcode` attribute, but the first has a `zip` attribute. By default, the root XML node (e.g., the `addresses` element) is contained in the Strudel collection `XMLRoot`. We use the terms nodes and objects interchangeably, but note that object does not denote a strictly typed value as it does in an object-oriented language or database.

Strudel was initially designed to query and manage only semistructured data. Most Web sites, however, integrate information from well-structured data sources, such as relational databases, flat files, or the output of ad-hoc shell commands. Strudel, therefore, also supports tuple-stream data, i.e., any data source that can be modeled by a finite stream of fixed-width records.

A StruQL query (middle of Fig. 1) is applied to semistructured and/or tuple-stream data sources, and its result is a *site graph*, which represents the site's content and structure and is completely in-

```
<! Postal addresses in XML>
<addresses id="alladdrs">
  <entry id="addr1">
    <name>AT&T Research</name>
    <address>
      <street>180 Park Ave.</street>
      <street>Bldg. 103</street>
      <city>Florham Park</city>
      <state>NJ</state>
      <zip>07932</zip>
    </address>
  </entry>
  <entry id="addr2">
    <name>INRIA Rodin</name>
    <address>
      <street>BP.105 Rocquencourt<street>
      <city>Le Chesnay</city>
      <postalcode>cedex 78153</postalcode>
      <country>France</country>
    </address>
  </entry>
</addresses>
```

Figure 2: Example of semistructured data

dependent of the output language. StruQL queries are compositional: a site graph is another example of semistructured data and can be queried by another StruQL query. A site graph is externalized on disk as an XML document.

To produce a browsable Web site, an *HTML template* is associated with each object in the site graph. Objects in a site graph may represent complete pages or page components. Usually, a template is associated with a collection of related objects, e.g., all account-page nodes. A template interleaves plain HTML text with STRUDEL-specific tagged expressions that access an object's attributes and format attributes' values. The template language is similar to other languages that separate presentation from content [5]. This technique simplifies the site programmer's task: he writes plain HTML extended with simple programmatic constructs, instead of a more complex scripting program that generates HTML. STRUDEL's generator (top of Fig. 1) evaluates the appropriate template for every object in a site graph. The resulting pages are the browsable Web site. Section 5 describes the template language in more detail.

2.1 Related Systems

Many commercial systems exist for designing and implementing Web sites. They include WYSIWYG HTML editors, tools for integrating database queries in individual Web pages, and model-driven, Web-site design systems. We focus on research systems and refer the reader to thorough reviews of site-development tools [11, 12]. Most of the problems associated with designing a Web site, such as modeling the site's content, specifying navigational structure, and customizing visual presentation, have been studied in the context of hypermedia systems, and many of the solutions for hypermedia systems are transferrable to Web-site design. The Autoweb [16], OOHDM [17], and Araneus [6] systems ascribe to a formal methodology of Web-site design, whose purpose is to isolate the various tasks of site design. Each system provides different tools, with varying levels of automation, to implement a design. Because their primary purpose is site design, neither Autoweb nor OOHDM-Web support querying or data integration. Like STRUDEL, Araneus separates data integration, site definition, and visual presentation, but it has two data models (one relational and one strictly typed graph) and two query languages, which cannot be composed naturally. We note that as an implementation tool, STRUDEL is complementary to site-design tools, because StruQL is well-suited to automatic generation and could be used as an implementation language for a variety of design systems.

Mawl [5] is a device-independent language for programming form-based services, which can be realized as Web applications or as interactive voice-response systems. Although STRUDEL's application is different, its separation of application logic from presentation and its template language are both inspired by Mawl.

The Document Object Model (DOM) [3] is a language-independent API for accessing HTML and XML documents, and the Extensible Stylesheet Language (XSL) [14] is a rule-based language for rendering a document in a markup language. These emerging standards are document centric, but may influence STRUDEL; e.g., a site graph could implement the DOM interface and possibly be rendered using XSL instead of STRUDEL's template language.

3 StruQL Query Language

We describe StruQL's core syntax by example, give an informal semantics, and describe query evaluation. In Sec. 4, we extend the StruQL with functions and forms. We illustrate StruQL's features

```
1    // Link root page to page of all accounts
2    link Root() -> "Accounts" -> AccountsPage()
3    // AccountsPage refers to each account in account database and its associated page
4    {  where (acct, name, street, city, state, zip) in SQL.query("AccountDB", "select acct ...")
5       link AccountsPage() -> "Info" -> Info(acct),
6            Info(acct) -> { "Acct" acct, "Name" name, "Street" street,
7                            "City" city, "State"  state, "Zip" zip,
8                            "AcctPage" AcctPage(acct) },
9            AcctPage(acct) -> "Info" -> Info(acct)
10
11     // AcctPage refers to non-zero usage records in the usage database.
12     { where (date, dom is int, intl is int) in SQL.query("UsageDB", "select date ...", acct)
13               dom + intl > 0
14       link  AcctPage(acct) -> "UsageData" -> UsageData(acct),
15             UsageData(acct) -> "Entry" -> UsageEntry(acct, date),
16             UsageEntry(acct, date) -> { "Date" date, "Total" (dom + intl) }
17     }
18     // Query postal database to determine possible aliases for account
19     { where XMLRoot{root},  root -> "addresses"."entry" -> addr,
20             addr -> { "name" alias, "address"."street" street1, "address"."zip" zip },
21             street1 = street
22        link  Info(acct) -> "Alias" -> alias
23     }
24   }
```

Figure 3: Fragment of site-definition query for `AcctPage` in HTN site

using the query in Fig. 3, which defines the account page in the HTN site. The site's data sources are two relational databases, of accounts and phone-call records, and one semistructured source of addresses. We focus on StruQL's declarativeness, i.e., a query specifies *what* the site's content and structure is, not *how* it is computed; its support for multiple data sources; and its controlled use of a general-purpose programming language (Java).

A StruQL query is a function that maps input-graph nodes and atomic values to a graph. A query's body is defined by the first EBNF grammar rule[2] in Fig. 4. The where clause is a conjunctive predicate expression. The link expressions link new nodes in the site graph, and collect expressions put nodes in the site graph's collections. *Node constructors* denote the OIDs of new nodes in the site graph. A predicate is a *collection expression*, a *regular-path expression*, an *atomic predicate*, or an *external-source expression*.

The query in Fig. 3 illustrates most of StruQL's features. The first clause on line 2 has an empty where clause, which is always true, so its associated link expression is always evaluated. `Root` is a node constructor that creates new object OIDs. By definition, a node constructor when applied to the same tuple of values always produces the same OID, so this expression creates two unique nodes, named `Root()` and `AccountsPage()`, and a link labeled `"Accounts"` between them.

The second clause (lines 4–9) contains an *external-source expression*. This expression binds the variables `acct`, `name`, etc. to the stream of tuples produced by the SQL query applied to the accounts database, `AccountDB`. For each binding of `acct`, the link expression on line 5 creates a new object, `Info(acct)`, and links `AccountsPage` to it. The expression on lines 6–8 copies all of `acct`'s attributes and values into the new `Info` object and links `Info` to its associated `AcctPage` object. Cycles between objects are permitted: line 9 links `AcctPage(acct)` back to its associated `Info` object. The nested clause (lines 12–17) is similar; it queries the usage database, which produces the domestic and international phone-call usage records for each `acct`. The where clause is satisfied when the sum of `dom` and `intl` is non-zero. The associated link clause groups usage entries by date in `UsageEntry(date, acct)` and stores the sum of `dom` and `intl`.

The last clause (lines 18–22) contains a *collection expression*, which binds a variable to every node in the specified collection, and *regular-path expressions*, which match arbitrary paths in an input

[2] { } delimit sequences, | separate alternatives, and [] delimit optional constructs

$$\begin{array}{ll} \mathit{Body} :- & \left[\text{where } \{\mathit{Predicate}\}\right] \\ & \left[\text{link } \{\mathit{NodeConstructor} \rightarrow \mathit{AttrExpr} \rightarrow \mathit{Term}\}\right] \\ & \left[\text{collect } \{\mathit{CollectionName}\{\mathit{NodeConstructor}\}\}\right] \\ & \{\{\mathit{Body}\}\} \end{array} \qquad \begin{array}{ll} \mathit{Predicate} :- & \mathit{CollectionName}\{\mathit{Var}\} \\ \mid & \mathit{Var} \rightarrow \mathit{RegularPathExpr} \rightarrow \mathit{Term} \\ \mid & \mathit{AtomicPredicate} \\ \mid & (\{\mathit{Var}\}) \text{ in } \mathit{ExtSourceExpr} \end{array}$$

Figure 4: *StruQL*'s EBNF grammar rules.

graph, e.g., an XML document. This clause is satisfied when there exists an object `addr` that is reachable from any member of the `XMLRoot` collection by a sequence of edges labeled `"addresses"."entry"`; `addr` must also have outgoing edges labeled `name` `address.street`, and `address.zip`. In general, 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. In addition to the concatenation (`.`) operator, R may contain the alternation (|) and (`*`) Kleene star operators.

The where clause on lines 19–22 is only satisfied when the values of `addr`'s `address.street` and `address.zip` attributes equal the values of `street` and `zip` bound in the first clause. In database parlance, this expression is a *join* on `street` and `street1`, or in logic-programming parlance, `street` and `street1` are unified. The two interpretations are equivalent. An explicit condition is not necessary: the join on `zip` is implicit, because it appears as the target of `address.zip`.

Figure 5 depicts a fragment of the site graph produced by the query in Fig. 3 applied to the sample relational data in Fig. 5 and the XML data in Fig. 2. The graph encodes the site's content and structure; e.g., the `Info` objects have links to account names and to account pages. The choice to realize objects as pages or as page components is delayed until HTML generation; our choice of node-constructor names (e.g., `AcctPage`) hints that some objects will be realized as pages, but this is not a requirement.

StruQL accesses user-defined methods using the Java reflection mechanism [4]. Any static Java class that implements StruQL's predicate, expression, or external-source interfaces is permissible. For example, StruQL provides the package `SQL` for accessing JDBC-compliant databases; it also provides a tuple-stream interface for flat files and shell commands and an interface to a Perl library for regular expression string matching.

StruQL supports the atomic types integer, float, string, date, and mime-content types such as URL, image, html, and postscript. By default, all values are interpreted as strings, but any variable can be annotated with an optional type, such as the `dom` and `intl` variables in Fig. 3(line 12). The query interpretor attempts to coerce values to the appropriate type at runtime. Although static typing is preferable, dynamic typing is necessary for StruQL, because the types of atomic values in data sources are usually unknown until query-evaluation time.

3.1 Semantics and Query Evaluation

A StruQL where clause is a *conjuctive query*. Conjunctive queries [2] are an important class of database queries, because they have several desirable properties. First, whenever the domain of a conjunctive query is finite, its range is also finite, which means that an evaluation of the query is guaranteed to terminate. Second, the *query equivalence* and *query containment* problems are decidable for conjunctive queries. Formally, given two queries Q_1 and Q_2, query equivalence decides for all inputs I, $Q_1(I) = Q_2(I)$, i.e., Q_1 and Q_2 compute the same result; query containment decides for all I, $Q_1(I) \subseteq Q_2(I)$. Query optimizers may rely on query equivalences and containments to eliminate redundant computations.

StruQL has an *active-domain* semantics, which means that the domain of a query, $\mathcal{D}$, is the union of the finite domains of the query's input graphs (i.e., object OIDs and atomic values), of its external sources, and of the set of constants that occur in the query. StruQL's semantics can be described informally in two stages. The *query stage* depends only on the where clause and produces all possible bindings of variables to values in $\mathcal{D}$ that satisfy all conditions in the clause. Its result is a relation R with one attribute for each variable; each tuple t in R satisfies the conditions. The *construction* stage

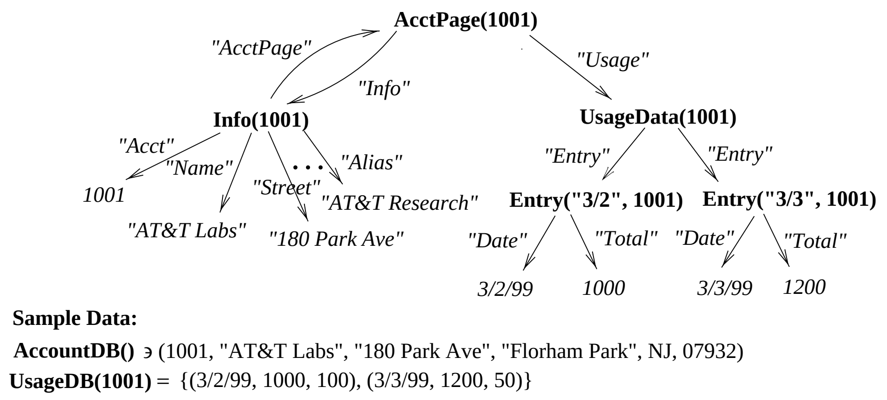

Figure 5: Fragment of site graph generated by query in Fig. 3

constructs the site graph by evaluating each link and collect expression once for every tuple t in R. The node constructor $N(v_1 \ldots v_n)$ denotes the object in the site graph whose OID is the value $N(\pi_{v_1 \ldots v_n}(t))$, i.e., the value of variables $v_1 \ldots v_n$ in t. Each link expression, $N(v) \rightarrow A \rightarrow N(w)$ creates an edge labeled A from $N(\pi_v(t))$ to the node denoted by $N(\pi_w(t))$. Each collect expression $C\{N(v)\}$ adds the node $N(\pi_v(t))$ to the collection C.

A StruQL query is evaluated by interpreting a *physical-operation tree*. STRUDEL's query-plan generator, like traditional query processors, translates a StruQL query into a physical-operation tree. Query-plan generation is similar to code generation in a compiler. The details of query-plan generation and strategies for efficient evaluation and optimization of StruQL are described elsewhere [9].

An important implication of StruQL's semantics is that a site graph is finite. In STRUDEL's first implementation, queries were evaluated completely and therefore materialized the entire site graph. We call this an *eager* evaluation strategy, which produces a *static* site graph. Eager evaluation is not always feasible or appropriate. For example, the query may range over gigabyte-sized data sources and therefore produce very large site graphs, or the site may depend on dynamically bound variables, e.g., inputs derived from a form. In addition to eager evaluation, STRUDEL now supports *lazy* evaluation, in which part of a site query is evaluated dynamically, e.g., at "click time"; a lazily evaluated query produces a *dynamic* site graph. Next, we introduce StruQL's functions, which modularize site-definition queries and are the minimal unit of query evaluation, and forms, which support dynamic binding of variables. After describing their semantics, we describe how flexible site-generation strategies can be implemented by combining eager and lazy evaluation of functions to produce sites that have both static and dynamic parts.

4 Extending StruQL with Functions

Our first applications of STRUDEL were small Web sites, like personal home pages, designed and maintained by one person. The HTN site was STRUDEL's first production application, and its site definition must be understandable and possibly extended by multiple people. HTN has several types of pages and several data sources, which made its definition in StruQL long and unwieldy.

To address these issues, we extended StruQL with functions. Functions are unusual in query languages[3], and adding them to StruQL is novel. StruQL's functions also differ from functions in general-purpose programming languages.

A StruQL function maps values in $\mathcal{D}$ (i.e., atomic values and input object OIDs) to a unique object (node) in the site graph and defines the entire subgraph accessible from that object. A function is defined by the EBNF grammar rule:

$$Function :- \textsf{fun}\ ID\ (\{Var\})\{Body\}$$

A function has a name (ID) and formal arguments (Var's), and its body consists of a StruQL query. The meaning of a function is that it constructs a subgraph and returns a reference to the graph's root. In the function body, the reserved node constructor `Result()` denotes the subgraph's root. There are two kinds of function calls, eager (`!f(x,y,...)`) and lazy (`?f(x,y,...)`), which we describe below. A *function call* is like a a

[3]While SQL defines *stored procedures*, these are not queries per se, but form a different language.

node constructor, i.e., it can occur in a collect expression, or as the target of a link expression:

```
link AcctEntry(acct) -> "AcctPage" ->
     ?acctPage(acct) // line 12 in Fig. 6
```

Here, `AcctEntry` is a node constructor and `acctPage` is a function call.

The subgraph returned by a function is disjoint from the graph constructed by the rest of the query and is connected to the rest only by edges to its root; e.g., the link expression above, constructs an edge `"AcctPage"` to the root of the subgraph returned by `acctPage(acct)`. Node constructors are locally scoped in a function's body, which guarantees that the nodes it constructs are disjoint from all others. For example, the body of the function `acctPage` in Fig. 6 contains the node constructors `Result`, `UsageData`, and `UsageEntry`, and these names are local to the function `acctPage`. Otherwise, function calls behave like node constructors, i.e., multiple calls to `acctPage(a)` with the same value for `a` produces exactly the same subgraph.

The `!(?)` prefix is a function-evaluation directive that specifies a callee function should be evaluated eagerly (lazily) when its caller function is evaluated. In Fig. 6, `acctInfo` is always evaluated eagerly; `reportPage` is evaluated lazily when called from `hotList`, but eagerly when called from `acctPage`. The user chooses a lazy or eager strategy based on efficiency considerations; the strategies' semantics are equivalent, i.e., both produce the same graph. We plan to generate strategies automatically and are experimenting with an engine that treats directives as "hints" that can be overridden. For example, an alternative strategy might evaluate all calls to `reportPage` lazily, because the page is accessed infrequently or because it is expensive to compute. In Sec. 4.2, we describe how one query can be evaluated using different site-generation strategies.

When a function is evaluated, a call to a lazily evaluated callee is replaced by a *closure* node, which encapsulates the information necessary to evaluate the callee. Calls to eagerly evaluated callees are replaced by the callee's result node. The HTML generator (Sec. 5) emits the appropriate HTML for either case.

Given FunStruQL's declarative semantics, functions differ fundamentally from those in other programming languages. All FunStruQL functions can be inlined, while preserving the query's semantics; in programming languages like C++ or ML, only non-recursive functions can be inlined. For example, the FunStruQL function:

```
fun f(x) { link Result() -> "self" -> !f(x)}
```

returns a node with a link to itself and is guaranteed to terminate. A call to `f`:

```
link Node(y) -> "call" -> !f(z)
```

would be inlined as:

```
link Node(y) -> "call" -> f_Result(z),
     f_Result(z) -> "self" -> f_Result(z)
```

Of course, inlining does not preserve the *operational* semantics of lazy functions, so our interpretor does not inline lazy function calls. In FunStruQL, function call arguments may be variables and constants, but not arbitrary expressions; this prevents an eager evaluation from resulting in a non-terminating computation, as it would in:

```
fun f(x) { link Result() -> "self" -> !f(x+1) }
```

Figure 6 contains the query for the HTN site. The function `hotList` (line 8) defines the top-level page that contains a list of those accounts in the `HotList` database that also occur in the account database, `AccountDB`. The wildcard _ ignores the value produced by an external source; in this case, `acct` must have *some* value in `AccountDB`, but the value itself is ignored. For each such account, a link is created to the corresponding `reportPage`, `acctPage`, and `acctInfo` nodes. The function `acctInfo` (line 17) computes general account information that is shared among several nodes and appears on multiple pages in the site. The `acctPage` function (line 23) links its result to the corresponding `reportPage` and accesses all the non-zero usage records from the `UsageDB`. It groups them by `date` and sums the `dom` (domestic) and `intl` (international) usage values. The `reportPage` function (line 33) lists all the reports known for the given account by querying an external reports database.

From a software-engineering perspective, this query has several desirable properties. Each function *identifies* the sources on which it depends, which makes it possible to reuse and modify the definition easily. A function also *encapsulates* a page, several pages, or a page component, making it possible to reuse parts of the site in different contexts; e.g., `acctInfo` is a page component contained in `acctPage` and `reportPage`.

```
// Root contains form to select hotList and one to select a specific account
{ where accttype in { "SmallBiz", "MiddleMarket", "ISP" }
  link Root() -> { (age, type) from HotListForm -> ?hotList(type, age),
                   (acct) from AcctForm ->  ?acctPage(acct) },
       Root() -> "AcctType" -> accttype
}
// hotList lists those accounts in HotList database that also occur in Account database
fun hotList(type, age) {
  where (acct, rank) IN HotList(type, age), (_, _, _, _, _) IN AccountDB(acct)
  link  Result() -> "Entry" -> AcctEntry(acct),
        AcctEntry(acct) -> { "ReportPage" ?reportPage(acct),
                             "AcctPage"   ?acctPage(acct),
                             "Info"       !acctInfo(acct),
                             "Rank"       rank }
}
// General account information is used in hotList, acctPage and reportPage
fun acctInfo(acct) {
  where (name, street, city, state, zip) IN AccountDB(acct)
  link  Result() -> { "Acct"   acct,   "Name" name, "Street" street,
                      "City" city, "State"  state,  "Zip" zip, "AcctPage" ?acctPage(acct)} }
// acctPage links to the acct's reportPage and to non-zero usage records
// in the usage database.
fun acctPage(acct) {
  link   Result() -> { "Info"        !acctInfo(acct),
                       "ReportPage"  !reportPage(acct) }
  { where (date, dom, intl) IN UsageDB(acct),  dom + intl > 0
    link  Result() -> "UsageData" -> UsageData(),
          UsageData() -> "Entry" -> UsageEntry(date),
          UsageEntry(date) -> { "Date" date, "Total" (dom + intl) } }
}
// reportPage lists all the reports known for the account by querying
// an external reports database
fun reportPage(acct) {
  link Result() -> "Info" -> !acctInfo(acct),
  { where (exec, date, comments) IN ReportsDB(acct)
    link Result() -> "Entry" -> ReportEntry(exec, date)
         ReportEntry(exec, date) -> { "AcctExec" exec,
                                      "Date"     date,
                                      "Comments" comments } }
}
```

Figure 6: Site-definition query for HTN site

4.1 Forms

Forms are an important feature of Web sites that cannot be expressed easily in a declarative query language, because they model sequential operations: get values from the user, then consume the values by constructing new graph nodes (i.e., pages). A FunStruQL *form* has the syntax:

$Form : -\ (\{Var\})$ **from** ID

A form may only occur on an edge in a `link` expression. Its effect is that, on traversing that edge, the user is prompted for the form variables' values, then the destination node is constructed; the latter must be a lazily evaluated function, because its arguments are not bound until runtime. For example:

```
link Root() -> (acct) from AcctForm ->
        ?acctPage(acct) // Fig. 6, line 4
```

defines the form `AcctForm` in the `Root` node. The HTML page for `Root()` must contain a form that binds `AcctForm`'s variables; this requirement is enforced by the HTML generator. Submission of the form's inputs by the user corresponds to a traversal of the edge `AcctForm` in the site graph; this binds the variable `acct` to the input value, and then the function `acctPage(acct)` is evaluated.

Conceptually, a query with forms defines an infinite graph, because the form's range of values can be infinite. This is not a problem, however, because only a finite portion of the site graph is ever expanded. Even though FunStruQL cannot restrict the set of values produced by a form to $\mathcal{D}$, the query's domain, we can check declaratively the validity of user's inputs. For example, the `type` of the `HotListForm` on line 3 should be restricted to one of the `AcctType` values. We could enforce that restriction in `hotList`:

```
fun hotList(type, age) {
 {where type in {"SmallBiz",
                "MiddleMarket", "ISP"},
        (acct, rank) IN HotList(type, age),
        (_, _, _, _, _) IN AccountDB(acct)
  link  // . . . from hotList in Fig. 5
 }
 { // Error clause
  where not(type in {"SmallBiz",
                    "MiddleMarket","ISP"})
  collect ErrorPage{Result()}
  link  Result() -> "BadArgument" -> "type",
        Result() -> "BadValue" -> type  }
}
```

The second `where` clause produces a node in the `ErrorPage` collection, which when realized in HTML, reports the invalid input to the user. It would be useful to have these declarative error clauses generated automatically given the range of an input variable.

4.2 Site-generation Strategies

The ability to support multiple site-generation strategies is especially important for data-intensive Web sites, in which the time to produce pages is non-uniform; e.g., some functions may submit expensive queries to an external source. In current practice, however, it is difficult for a site developer to support more than one site-generation strategy. So by default, most sites are generated either dynamically or statically.

Strudel already supports site-generation strategies defined explicitly in the query, but we would also like to support those defined automatically by a profile-driven site optimizer. For example, when account reps begin work each day, they scan the hot lists for new accounts in their market segments. The first pages accessed in the site can be identified by examining the HTTP-server trace logs, because the CGI-bin calls encode the names of the FunStruQL functions and their argument values. Frequently accessed pages can be precomputed by simply adding clauses to the anonymous function; for example, this clause precomputes all new, ISP accounts and can easily be generated automatically:

```
link // Precompute new, ISP accounts.
Precompute() ->"HotList"->!hotList("ISP","new")
```

The precomputed pages are cached and immediately available when the user requests them.

Some strategies, however, cannot be inferred automatically. For example, after accessing the appropriate hot list, an account rep scans through the reports for 10-20 of the highest risk accounts, i.e., those with low rank. We could express this strategy by the clause:

```
// Site-generation strategy: precompute
// reports of "new", "ISP" accounts with low rank
where (acct, rank) IN HotList("ISP", "new"),
       (_, _, _, _, _) IN AccountDB(acct),
       rank < 20
link
 Precompute() -> "ReportPage" -> !reportPage(acct)
```

We cannot infer this clause automatically from server logs, because the logs encode the account numbers of the selected accounts, and on any particular day, a *different* set of accounts has the lowest rank. In this case, the strategy might have to be specified by the site developer. Note that FunStruQL's declarativeness makes it easy for a site optimizer or a developer to specify strategies.

Functions help modularize a query, but they also can introduce redundant computations. FunStruQL's semantics, however, makes it possible to identify and eliminate these computations automatically. For example, the `hotList` function (Fig. 6, line 9) checks that every account in the hot list exists in the account database and then calls `acctInfo`, which queries the same source. When called from `hotList`, `acctInfo`'s query is redundant, but it is not redundant when called from other functions. We can prove that when called from `hotList`, the result of `acctInfo`'s `where` clause is *contained* in `hotList`'s result, and we could optimize `hotList` by inlining `acctInfo` and eliminating the extra query to `AccountDB`:

```
// Optimization: avoid recomputation of acctInfo
fun hotList(type, age) {
 where (acct, rank) IN HotList(type, age),
       (name, street, city, state, zip)
            IN AccountDB(acct)
 link Result() -> "Entry" -> AcctEntry(acct),
      AcctEntry(acct) ->
         { "ReportPage" -> ?reportPage(acct),
           "AcctPage"   -> ?acctPage(acct),
           "Info"       -> AcctInfo(acct),
           "Rank"       -> rank },
      AcctInfo(acct) ->
         { "Acct" acct, "Name" name,
           "Street" street, "City" city,
           "State"  state,  "Zip" zip } }
```

As with any program optimization, an important problem is deciding *where* to apply optimizations. Although we do not address this problem here, we note that FunStruQL's declarative semantics simplify implementation of query optimizations.

5 Template Language

One premise of Strudel's design is that a site's HTML rendering is separable from the site's content and structure. Strudel's template language allows the user to specify a site's HTML rendering. A template is a function that maps an object in a site graph to an HTML value. The template's expressions produce HTML values, which are concatenated to produce its result, and are defined by the EBNF rules in Fig. 7. Plain HTML text, the format expression (`sfmt`), conditional expression (`sif`), enumeration expression (`sfor`), and form expression (`sform`) are sufficient for emitting a site graph in HTML. An attribute expression, e.g., `Info.Acct`, denotes the set of objects reachable by edges labeled with the given attributes. An attribute expression implicitly refers to the template's object argument, named `this`, but can refer explicitly to any object variable, e.g., `@this.Info.Acct`. Sometimes more general computation is necessary during HTML generation; the `sjava` construct provides an "escape" into Java, which permits the evaluation of arbitrary Java code.

For each object in a site graph, Strudel's generator applies the appropriate template to the object to produce its HTML value. Each object in a site graph has a user-specified *generation mode*: *page* or *page component*; all leaf objects, i.e., atomic values, are page components. Figure 8 contains fragments of the templates for the `Root`, `acctInfo`, and `acctPage` objects in the HTN site. The `Root` and all `acctPage` and `reportPage` objects are realized as pages, and all `acctInfo` objects as page components.

The format expression maps an object to an HTML value. In the `acctInfo` template (Fig.8), `<sfmt Name>`, refers to the atomic value reachable by the attribute expression `@this.Name`, and is replaced by its HTML value, a string. Format expressions are concise, because the generator uses type-specific rules to determine an object's HTML value. For most atomic values, the object's HTML value is converted to a string. For some atomic values, e.g., those with type PostScript, the generator produces a link to its value.

An internal object's generation mode determines how it is formatted. In `acctPage`'s template, `<sfmt Info>`, always refers to an `acctInfo` object `a`, which is a page component, so it is replaced by `a`'s HTML value, but the expression `<sfmt ReportPage link="All Reports">` refers to a `reportPage` object, which is a page, so it is replaced by a link to the appropriate page; the `link` directive specifies the link's tag text.

Some internal objects are *closures*, which represent lazily evaluated functions. In `acctInfo`'s template, `<sfmt AcctPage link=Acct>`, refers to a closure for the lazily evaluated function `acctPage` (as defined in the query in Fig. 6). Its result is a page object, so the generator emits a link that contains a

$$\begin{array}{lll}
\mathit{Template} & :- & \{\mathit{Body}\} \\
\mathit{Body} & :- & \mathit{PlainHTMLText} \\
 & | & <\textbf{sfmt}\ \mathit{AttrExpr}[\textbf{link} = \mathit{AttrExpr} | \mathit{String}]> \\
 & | & <\textbf{sif}\ \mathit{CondExpr}>\mathit{Body}</\textbf{sif}> \\
 & | & <\textbf{sfor}\ \mathit{ID}\ \textbf{in}\ \mathit{AttrExpr}[\textbf{order} = (\textbf{ascend}|\textbf{descend})\ \textbf{key} = \mathit{AttrExpr}]>\mathit{Body}</\textbf{sfor}> \\
 & | & <\textbf{sform}\ \mathit{AttrExpr}>\mathit{Body}</\textbf{sform}> \\
 & | & <\textbf{sjava}>\mathit{JavaCode}</\textbf{sjava}> \\
\mathit{AttrExpr} & :- & [@\mathit{Var}\textbf{.}]\mathit{Attribute}\{\textbf{.}\mathit{Attribute}\}
\end{array}$$

Figure 7: EBNF Rules for the HTML templates.

Root template:

```
<sform HotListForm>
 Choose account age and type:
 Age: <sinput type="text" name="age" value="old">
 Account type: <sinput type="select" name="type">
 <select>
 <sfor t in AcctType>
    <option value="<sfmt @t>"><sfmt @t>
 </sfor>
 </select>
</sform>
```

acctInfo template:

```
<h1>Account #<sfmt AcctPage link=Acct></h1>
<sfmt Name>: <sfmt Street>, <sfmt City>
<sif PostalCode><sfmt PostalCode>
<selse><sfmt Zip>
</sif>
```

acctPage template:

```
<html><sfmt Info><hr>
<sfmt ReportPage link="All Reports">
<table><tr><td>Date</td><td>Total</td></tr>
<sfor e in UsageData.Entry
       order=descend key=Date>
  <tr><td><sfmt @e.Date></td>
      <td><sfmt @e.Total></td></tr>
</sfor>
</table>
</html>
```

Figure 8: Templates for `Root`, `acctInfo`, and `acctPage` objects of HTN site

STRUDEL-specific, CGI-bin expression that encodes the closure function's name and its argument values. When the link is selected at runtime, STRUDEL evaluates the appropriate function and produces the result page. If a closure object produces a page component, the generator applies the closure to produce the object then applies its template to produce its HTML. Note that template expressions are independent of the site-generation strategy, which means the template writer does not have to know how an object is produced. The generator emits the appropriate HTML code whether an object is the result of an eagerly or a lazily evaluated function.

The `sif` expression evaluates a conditional expression and then evaluates the appropriate branch. For example, in the `acctInfo` template:

```
<sif PostalCode> <sfmt PostalCode>
<selse> <sfmt Zip> </sif>
```

tests for a `PostalCode` attribute and emits its value if it exists, otherwise it emits the `Zip` attribute's value.

Objects can have multiple instances of the same attribute, e.g., `acctPage` objects have multiple `UsageData.Entry` attributes. The `sfor` expression binds an object variable to each object denoted by its attribute expression and evaluates its body for each binding. In the `acctPage` template,

```
<sfor e in UsageData.Entry order=descend key=Date>
  <sfmt @e.Date> <sfmt @e.Total>
</sfor>
```

binds `e` to each value of the attribute expression `UsageData.Entry` and emits `e`'s `Date` and `Total` values. The `order` directive sorts objects in either lexicographically increasing or decreasing order; if the objects are internal, the optional `key` value specifies the attribute that should be used as the sort key. The `sfor` expression above orders the `UsageData.Entry` objects in descending order by their `Date` attribute.

The attribute expression of a `sform` must refer to a *form* object. A form object has free variables, and, like a closure, a target function and some bound variables. In the `Root` template, for example, the `<sform HotListForm>` expression refers to the `HotListForm` object. All of a form's free variables must be bound by `sinput` expressions within the body of the `sform`. In this example, `HotListForm`'s variables `age` and `type` are bound; the possible options for `type`'s value are enumerated by the `sfor` expression. As with closures, the template writer need not know how the target function is evaluated. The generator emits the appropriate "action" value for the HTML form, which includes the target function and its bound variables. Currently, the check that a form's free variables are bound is done dynamically during HTML generation.

Although HTML is the standard output language for a site graph, it is not the only one. STRUDEL can emit a site graph in XML and fewer than 100 lines of STRUDEL's generator are HTML-specific, so other markup languages could be supported with minimal changes to the generator.

6 Evaluation and Discussion

As described in Sec. 1.1, the original HTN site was completely re-implemented using STRUDEL and Daytona, a relational database management system. We compare the total number of files and total number of non-empty, non-comment lines of code for each implementation. Reducing the total line count is not a definitive measure of improvement, but it does indicate the relative effort required for each implementation. Table 1 compares the two implementations. Each source-code file was categorized as primarily site-definition code, HTML-template code, or general-purpose Java code. In the STRUDEL implementation, 66% of the code is devoted to page presentation, but less than 30% is required to define the site. This is encouraging, because the site-definition query contains the potentially reusable part of the specification and is the first and only component that a user would read to understand the site's definition.

In the original implementation, 75% of the code is devoted to site definition, but more importantly, the code to access data, to define site structure, and to emit HTML code is interleaved, making it difficult to modify or extend. Overall, the STRUDEL implementation is 1.6 times smaller than the original implementation, but if we compare only the code for site-definition, it is more than 4 times smaller. Also, the STRUDEL definition is encapsulated in one file, whereas the original definition was distributed over 23 files.

	Implementations			
	STRUDEL		Original	
Type of code	# lines	# files	# lines	# files
Site definition	**291**	1	**1198**	23
Templates	673	11	42	1
Java code	41		392	1
Total	**1005**	12	**1632**	26

Table 1: Comparison of HTN site implementations

Unlike the original implementation, the STRUDEL implementation supports flexible site-generation strategies. For example, we implemented by hand some simple strategies, similar to those in Sec. 4.2: precompute frequently accessed hot lists and report pages. These added less than 10 lines to the anonymous function, and in the best cases, reduced page-generation time from 12 seconds to less than 2 seconds. The strategies *extend* the original query with hand-coded optimization rules. Our next challenge is to generate these strategies automatically. HTTP-server trace logs and STRUDEL profiling statistics can provide useful optimization information.

Our general design strategy was to focus on the hardest problems of creating data-intensive Web sites: accessing and integrating data and building the site's content and structure. Our first insight was that these problems are best solved by a declarative query language. Our second insight was that unlike ad-hoc queries in traditional query languages, a StruQL query is also a software artifact, which must be extensible and reusable. We extended StruQL with functions in a way that preserved the simple semantics of StruQL, but that better enabled STRUDEL to support dynamic sites and flexible site-generation strategies.

Overall, FunStruQL's simple, declarative semantics make the language easy to understand and more importantly, easy to analyze. We have already mentioned some unexpected benefits, e.g., declarative specification of error clauses. An important problem we have not addressed yet is extending FunStruQL with an *update semantics*, i.e., a formalism for specifying updates to a query's domain, and a syntax for specifying updates. Given an update semantics, STRUDEL could support *incremental update* of a site, i.e., identify those parts of the site graph effected by an update and recompute automatically the pages effected.

One common criticism of FunStruQL in particular, and other domain-specific languages in general, is they perpetuate the "Tower of Babel", requiring the user to learn a new language when well-known programming languages can solve the problem at hand. Our response is that FunStruQL's long-term benefits should outweigh the short-term cost of learning the language. Site definitions in FunStruQL are self-documenting and shorter than the equivalent scripting code, making it easier to modify and reuse them immediately. The HTN site substantiates many of FunStruQL's benefits, but we expect that applying it to other sites will reveal other opportunities for improvement.

Remarks. STRUDEL is available from `http://www.research.att.com/sw/tools/strudel`. We thank Sandra Sudarsky for her contributions to STRUDEL's implementation.

References

[1] S. Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Weseley, 1995.

[3] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model level 1.0 specification. Technical Report REC-DOM-Level-1-19981001, World Wide Web Consortium, Oct. 1998.

[4] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition.* Addison-Wesley, Dec. 1997.

[5] 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.

[6] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive web sites. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 436–450, Valencia, Spain, 1998.

[7] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, February 1998.

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

[9] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experiences with a web-site management system. In *SIGMOD*, Seattle, Wash., June 1998.

[10] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Verifying integrity constraints on web sites. In *IJCAI*, 1999.

[11] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3), Sept. 1998.

[12] P. Fraternali. Tools and approaches for developing data-intensive web applications: a survey. *ACM Computing Surveys*, Sept. 1999.

[13] R. Greer. Daytona. *Proceedings of the SIGMOD International Conference on Management of Data*, June 1999.

[14] X. W. Group. Extensible stylesheet language (xsl). Technical Report WD-xsl-19981216, World Wide Web Consortium, Dec. 1998.

[15] J. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[16] P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, 1998.

[17] D. Schwabe and G. Rossi. An object oriented approach to web-based application design. *Theory and Practice of Object Systems, Special Issue on the Internet*, 4(4):207–225, 1998.