The Debye Environment for Web Data Management

The Debye environment lets users extract and manage semistructured data available from Web sources, using an extended form of nested tables as its fundamental paradigm.

Currently, the Web contains a large amount of interesting data implicitly available on pages at various sites, including digital libraries and online stores. Researchers regard these data-rich pages as “data containers,” because they contain useful, semistructured data. Such data is not readily available through conventional Web search tools, however, as it is typically identifiable only indirectly through visual clues such as colors, fonts, bullets, and indentations. Further, the underlying flexibility of both the content and format creates structural variations and irregularities that challenge traditional data management systems. Even though data structuring standards such as XML are likely to gain in popularity, that fact does not address the existing (and still growing) volume of semistructured Web data available, for instance, on HTML pages.

To contend with this challenge, we’ve developed the Data Extraction by Example (Debye) approach to managing semistructured Web data. Our goal with Debye is to provide tools for extracting this data from its original sources, and logically representing the data in a format that permits further manipulation. Here, we discuss our approach, focusing not on the technical novelties (which we’ve presented elsewhere1-4), but rather on how Debye uses extended nested tables to support solutions to several Web data management problems.

Currently, the Debye tools are fully implemented as prototypes for applications that use data from heterogeneous Web sources. For instance, developers have recently used Debye to build a repository that integrates data from the digital libraries of several universities.

Debye: An Introduction

To present Debye’s features, we’ll use the example in Figure 1: an Amazon.com Web page containing semistructured data that appeared in response to the query, “Paul McCartney.” To monitor new releas-
es, changing prices, and new auction offerings on products related to Paul McCartney, users typically have to use manual approaches. These might involve repeatedly visiting the site, issuing queries, recording relevant data, and then comparing it with old data. As the volume of targeted products grows, so too does the potentially unbearable effort required to keep up.

Our approach offers an alternative: extracting target data from pages of interest and storing it locally in nested tables. Figure 2 shows a nested table containing data from the page in Figure 1. At the Amazon site, each department offers distinct product information; to simplify our example, we consider only products from Popular Music, Books, and Auction. For each department, the table’s Info column stores an internal table with a distinct structure. Once such a table is built, users can perform sophisticated queries and store the extracted data in a relational database for further processing.

**Debye Environment Overview**

Figure 3 (next page) shows the Debye tools and how they are related to each other. For the data extraction task, the Debye environment features a graphical user interface that lets users assemble example objects by cutting and pasting data values from the target source page. From these example objects, the GUI generates an object-extraction pattern (OE pattern), which indicates how to locate and structure the objects implicitly present in target pages. The OE pattern is fed to a generic extractor; it also serves as a guide for extracting new objects from pages similar to the sample page. The extractor outputs the extracted objects in an XML-based format, creating a Debye textual object repository (DTOR).

The Debye environment also features a user-independent example generator. The example generator matches data values from a sample Web page with those in an existing data repository that contains objects from the Web page’s application domain. It then uses these values to assemble example objects. Based on the examples generated, the example generator outputs an OE pattern (in the same way as the GUI) for the extractor to use. Using this example generator makes the OE pattern generation process both resilient (immune to changes in the source page’s format) and adaptive (capable of working with pages from distinct sources in the same application domain).

One of the advantages of using nested tables for representing the extracted data is that they let us extend well-known query operations for nested tables to deal with internal variations as defined by the Debye data model. We implemented these operations in a graphical query interface suitable for semistructured Web data. It combines features of the Query by Example (QBE) language with typical features of query languages for semistructured data. In particular, the interface provides the structure of the data as a nested table “skeleton” so that users do not have to uncover the data’s structure by themselves.

Similarly, the underlying tabular structure of the data we manipulate simplifies the task of storing it into relational databases. Researchers have explored relational databases as an alterna-
Database Technology on the Web

tive to storing semistructured data because they can efficiently manage huge volumes of data.\(^{5,6}\)

Therefore, taking advantage of our underlying data model, we implemented a mechanism for storing and retrieving semistructured data in relational databases, which we call the Debye data storage manager.

**Debye Data Model**

We base our Debye approach on the users’ perception of the data’s structure. Debye captures this perception using an extended form of nested tables, in which a column within the table allows two or more distinct substructures (see Figure 2).

Although the resulting representation is not as expressive as general semistructured data models (such as the Object Exchange Model\(^{7}\) or XML), it can adequately represent typical semistructured data available on data-rich Web pages, as our experiments\(^{1}\) and a comparison between Figures 1 and 2 show.

To characterize these ideas more precisely, we define a table scheme using the following notation:

\[
\tau = \left( C_1 : \tau^1_1 ; \ldots ; \tau^1_{n_1} , \ldots , C_m : \tau^m_1 ; \ldots ; \tau^m_{n_m} \right)
\]

where \(m \geq 2\), \(n_k \geq 1\), \(k = 1, \ldots, m\), \(C_j\) is a column, and \(\tau^j_{n_j}\) denotes exactly one of the following: an atomic value, represented by \(\text{atom}\); a set of atomic values, represented by \(\{\text{atom}\}\); or a table scheme.

To simplify the notation, if \(n_j = 1\), we can use \(C_j : \tau^j_1\) instead of \(C_j : [\tau^j_1]\). Also, if \(C_j : \text{atom}\), we call \(C_j\) an atomic type.

Intuitively, a table scheme describes a nested table structure in which a column \(C_j\) can store, in distinct tuples, values or objects with a distinct structure. Possible object structures are given by the alternatives, \(\tau^j_1 ; \ldots ; \tau^j_{n_j}\), which can be either atomic values, lists of atomic values, or other nested tables. As Figure 2 shows, for example, we can describe the object structure implicitly present in Figure 1 using the following table scheme:

\[
\text{ProductList} = (\text{Store:atom, Info: } [(\text{Title:atom, Artist:atom, AudioType:atom}); (\text{Title:atom, Authors:atom, BookType:atom});(\text{Item:atom, Bid:atom, Time:atom})]).
\]

**Data Extraction**

According to our approach, to manage data available in data-rich Web pages, we must first translate it from its original textual format to a suitable processing format, such as XML or relational tables. This is usually accomplished by wrappers. We represent a wrapper generation as follows: Given a Web data source \(S\) containing a set of pages \(T\), determine a mapping \(w\) that can populate a data repository with a set of objects (data items) extracted from \(T\).

In general, the mapping \(w\) is a set of rules or text patterns that recognizes — among other uninteresting pieces of text — attribute values for target objects, associating them with appropriate semantics. Thus, a wrapper is an implementation of the mapping \(w\), which corresponds to an OE pattern in our approach.

**Debye GUI**

To help users specify example objects, the Debye environment includes a GUI based on nested tables. Figure 4 shows a snapshot of an example specification session for the sample page presented in Figure 1.

To assemble an example, users mark pieces of data from the Source window page and copy them into the columns of a table in the Example window. They do this separately for each piece of.
text. The tool also provides several column operations not shown in the figure, including Insert, Remove, Rename, Group, and Split. These operations let users build a nested table that embeds the example object’s structure. In Figure 4, the user has assembled three distinct rows, each one corresponding to a product type. The resulting nested table resembles the one in Figure 2. Once users have specified example objects, they activate the Pattern button (in the function bar) to build the corresponding OE pattern for the extractor.

**OE Pattern Generation**

In Debye, the GUI generates OE patterns, based on a nested table assembled by users, which captures the target objects’ structure. This feature is key to our approach: It lets users implicitly inform Debye of the object’s syntactic context and structure through examples. Because some target pages, such as the one in Figure 1, contain non-homogeneous objects, users would have to provide more than one example. A user might specify, for example, each possible structural variation for a Store instance. In practice, few examples are needed.1

Given the user-provided examples, Debye’s GUI generates the corresponding OE pattern, which essentially encodes two kinds of information required to guide the data-extraction process:

- the target object’s structure, in the form of a table scheme; and
- the textual surroundings — such as markups, symbols, and keywords — of the atomic values of the object’s attributes. We refer to these values as attribute-value pairs.

More precisely, an OE pattern is a pair, \( \langle \tau, \rho \rangle \), in which \( \tau \) is a table scheme and \( \rho = \langle a_1, \pi_1 \rangle, ..., \langle a_n, \pi_n \rangle (n \geq 1) \) is a list of attribute-value pair patterns (AVP patterns). In these pairs, \( a_i \) is an atomic type that is part of \( \tau \), and \( \pi_i \) is a regular expression for finding values of type \( a_i \) in the input pages.

To generate regular expressions, Debye generalizes the surroundings of each value of \( a_i \) in the example, such that each \( \pi_i \) can recognize all values that share a common textual context. For each atomic type composing the table scheme ProductList, for instance, at least one AVP pattern will describe the context in which the type values occur in the target pages. We discuss how Debye generates AVP patterns elsewhere.1

**Extraction Strategy**

Once the Debye GUI generates the OE pattern, the Debye extractor reads it and parses its rules to guide the extraction process for input pages. Thus, we regard the extractor as a sort of configurable generic wrapper. More precisely, we regard an OE pattern as a grammar and the extractor as a parser.

Given an OE pattern and a set of input pages, the target data is extracted in an efficient, bottom-up procedure. The extractor first recognizes and extracts atomic components, and then uses them to assemble the complete object through a bottom-up composition operation. For example, let’s say we’re interested in obtaining object instances of
the table-scheme type:

\[
\text{Author} : (\text{Name} : \text{atom}, \{\text{Book} : (\text{Title} : \text{atom}, \text{Price} : \text{atom})\})
\]

That is, we want an author’s name, along with a list of her books and their price. Let’s further suppose the extraction procedure receives as input the OE pattern \( \langle \tau, \rho \rangle \), for which

\[
\rho = \{\text{Name}, \pi_{\text{Name}}\}, \{\text{Title}, \pi_{\text{Title}}\}, \{\text{Price}, \pi_{\text{Price}}\}
\]

Figure 5 shows the extraction process steps.

In the first step, the extractor uses the AVP patterns to find instances of values of attributes such as Name, Title, and Price. The result is a list of attribute-value pairs (the lowest row of circles in Figure 5) ordered by a label, \( l_i \), which indicates the source-page position of each pair’s corresponding string.

In the second step, the extractor combines contiguous pairs of Title and Price values to form Book instances, which are labeled with the smallest \( l_i \) (the second lowest row in Figure 5). Note that the bottom-up procedure builds the Book instance labeled \( l_i \) with distinctly ordered components. Also, the Book instance \( l_0 \) is missing its Price component. Our bottom-up procedure lets us deal with situations in which it assembles complex objects using only the extracted components, regardless of their order.

Next, the bottom-up procedure takes the Book instances related to the Author in a list, \( \{\text{Book}\} \). It then combines each list with an instance of Name (previously extracted), to assemble Author instances.

**Automatic Example Generation**

Traditional example-based approaches for wrapper generation share two undesirable characteristics. First, changes in the target Web page’s format or layout can spoil the wrapper’s function if the textual contexts that it uses to recognize data of interest change. Second, wrappers are specific to a single Web data source. Thus, a wrapper for a given Web source cannot be used for another, even if they share an application domain. These limitations are due to the fact that wrappers are generated using examples provided by users from a sample page of a specific Web source.

To overcome these limitations, we devised an automatic example-generator module that provides a user-independent bootstrapping method for example-based Web data extraction. The example generator allows the creation of wrappers that are both resilient and adaptive. Consider, for instance, a wrapper \( w \) generated using a set of example objects \( E \) taken from sample pages \( T_0 \subset T \), where \( T \) is a page set taken from a Web source \( S \).

We define a data-extraction process as **resilient** if the extraction level remains the same for any page set \( T' \) composed of pages that are new versions of pages in \( T \) taken from the same Web source \( S \). (We assume that the pages’ formatting features and layout have changed, but that the content — the target objects — remains the same.) We define the process as **adaptive** if its extraction level remains approximately the same for any page set \( U \), composed of pages taken from a Web source \( S' \), which is distinct from \( S \) but has objects that share an application domain with objects in \( T \).

Figure 6 shows the framework for automatically creating examples for the wrapper-generation process. Suppose a user wants to generate a wrapper for extracting data from Web source \( S \) that contains objects (data) on a given domain, such as music. We assume the existence of a bootstrapping data repository \( R_0 \) — the source repository — which contains a set of objects belonging to the same domain as \( S \). First, the data matcher tries to recognize, in \( S \)’s sample pages, matches for the attribute values of \( R_0 \)’s objects. The result is a set of candidate attribute-value pairs taken from the sample page text. Next, the example assembler module selects those attribute-value pairs that are likely (according to various heuristics) to compose objects in the \( S \) domain, and uses the pairs to assemble example objects. The entire process is guided by the structure of \( R_0 \) objects, which follows the Debye data model.

One of the most challenging problems we faced here was how to automatically recognize the intersection of \( R_0 \) data with the target sample pages. For example, \( R_0 \) might include the string “Paul McCartney” as an artist name value, but the name might appear in a page as the string “McCartney, P.” Although these two strings are not the same, they refer to the same person and...
Related Work in Managing Web Data

Researchers have proposed many systems and environments for Web data management. Most such systems deploy graph-based formalisms to represent the structure and contents of Web sites and pages. Although this has yielded elegant solutions to the problems of extracting, querying, and integrating Web data, practical use unfortunately requires that users have some understanding of the formalism. More recently, systems for directly managing XML data have also been proposed and implemented.

Web Data Extraction

For Web data extraction, researchers have recently proposed several approaches for wrapper generation. As we discuss elsewhere, these approaches are based on techniques such as natural language processing, machine learning, data modeling, and ontologies. Other approaches rely on HTML documents’ inherent structural features to extract data. Although there are similarities between Debye and these approaches, our use of nested tables as the underlying data model distinguishes our approach in many respects. In particular, this gives users a simple and intuitive metaphor for data modeling and provides a flexible extraction process.

In other research, only David Embley and his colleagues use an approach that generates wrappers that are inherently resilient and adaptive. With the RoadRunner system, the wrapper-generation process is fully automatic, and thus resilience and adaptiveness are not an issue. Our own method for extending Debye to provide resilience and adaptiveness is discussed elsewhere.

Querying Web Data

Currently, one of the most active research topics in Web data management is the development of query languages and tools for semistructured Web data. Initially, researchers proposed languages such as Lorel for querying data represented using specific data models. Later, dozens of XML-targeted languages began to appear. A standard XML query language, XQuery, is now a W3C working draft.

Storing Semistructured Data

There are three basic alternatives for using relational database management systems to store semistructured data, addressing all table levels.

Query Interface

Debye’s query interface provides four query operations — selection, projection, nest, and unnest — which are extensions of the recursive query operations for regular nested tables.

- The selection operation lets users specify selection conditions at any nesting level. A selection condition is a Boolean expression defined over either a column’s value or the existence of an internal table’s column (a structural condition). The query interface evaluates selection conditions throughout nested tables, addressing all table levels.
- The projection operation is similar to the one defined in relational algebra. Essentially, it horizontally reduces a table by keeping only user-specified columns. To deal with the Web’s semistructured data, the operation recursively verifies the existence of each user-specified column in all table rows.
- The nest operation has two distinct semantics. When applied to a single column, it groups a column’s values with equal values in other columns. When users apply the nest operation to a set of columns, it creates a new internal table that groups the column values and, consequently, generates a new level of nesting. The query interface recursively applies the nest operation to the entire table.
- The unnest operation is the inverse of the nest
operation. It also has two distinct semantics, and must always be applied to a column whose contents are either a list of atoms or an internal table. When users apply it to a list of atoms, it “ungroups” its elements — that is, it splits them into different table rows. When they apply it to an internal table, it eliminates a nesting level. As with the nest operation, the unnest operation is recursive.

The query interface implements these operations, combining features of the QBE language with features of query languages for semistructured data, such as type coercion and path expressions. Our interface is simpler to use, however, because it takes advantage of the fact that the extraction process uncover the data repository’s structure and that the interface gives users a table “skeleton” that graphically describes possible variations in the repository’s data.

To illustrate the query formulation process, we use an example query of a repository generated from Amazon.com pages, much like the one in Figure 1. Suppose the user’s query was: “List the title and type of the books written by John Grisham, and rearrange the result by nesting the values of the column BookType.” To formulate this query, the user first selects the target repository. This immediately produces a table skeleton that shows the stored data’s structure (see Figure 7a).

The user then specifies the selection conditions on the columns Store and Authors to retrieve only books written by John Grisham. As the Figure shows, the user specified the condition “~” on Authors, which denotes approximate comparison by pattern matching. To show only the Title and BookType, the user specifies a projection operation, clicking on the corresponding column headers — indicated in Figure 7a by: (p).

Figure 7b shows an excerpt of the query result. To satisfy the query specification, the user must rearrange the result by nesting the values of the column BookType.” To formulate this query, the user first selects the target repository. This immediately produces a table skeleton that shows the stored data’s structure (see Figure 7a).

The user then specifies the selection conditions on the columns Store and Authors to retrieve only books written by John Grisham. As the Figure shows, the user specified the condition “~” on Authors, which denotes approximate comparison by pattern matching. To show only the Title and BookType, the user specifies a projection operation, clicking on the corresponding column headers — indicated in Figure 7a by: (p).

Figure 7b shows an excerpt of the query result. To satisfy the query specification, the user must rearrange the result by nesting the values of the column BookType.” To formulate this query, the user first selects the target repository. This immediately produces a table skeleton that shows the stored data’s structure (see Figure 7a).
The main challenge for storing semistructured Web data in relational databases is the relational model's lack of expressiveness in representing irregular hierarchical data. To solve this, we developed a mapping method that preserves the data's multilevel structure using standard relational-model resources.4

Figure 8. Mapping method. The method preserves the data's multilevel structure using two procedures: Map-Table and Map-Column.
store and then manipulate semistructured Web data:

- Store the entire repository or document in a single relation attribute (as in several commercial DBMSs, such as Oracle and DB2);
- Treat repository objects as graphs and represent their nodes and arcs using relations; or
- Map repository objects to a corresponding relational database.

Among these alternatives, only the last lets users exploit relational DBMS features, such as querying optimization and concurrency control. Given this, we adopted this approach in Debye. Mapping arbitrary, semistructured objects to relational databases may require revealing a regular structure that describes them, a problem identified as NP-complete. Storing Debye objects is much simpler, however, because we take advantage of user modeling to implicitly identify such regular structures.

**References**


**Figure 9.** Resulting relational database. In this figure, relations Info1, Info2, and Info3 represent the three alternative structures for the contents of the Info column.
First, when compared with textual repositories, storing data in a relational database requires additional disk space (mainly due to index structures). Second, the system requires additional time to assemble the components of objects that were split among the generated relations whenever it had to totally or partially recover an original repository.

**Future Work**

Currently, the Debye environment components are fully operational as prototypes and are being used experimentally in Web data management applications. An important issue we have not addressed here is how to automatically obtain the Web pages for data extraction. To this end, we’ve designed and implemented a tool to assist users in generating agents to collect Web pages that contain relevant data, possibly produced dynamically as the result of form submissions. We will add this tool to the Debye environment soon.

As for future work, we are currently addressing the problem of Web data integration. Although the Debye data model provides a suitable modeling framework for properly “merging” the objects to be integrated, we’ve yet to solve the problem of establishing identity between objects. This is now one of our main research directions.

**Acknowledgments**

This work was partially supported by the Information Systems on Mobile Computing Environments (SIAM) Project (grant number 76.97.1016.00) and by the Brazil National Council for Scientific and Technological Development (CNPq) (grant number 467775/00-1).

**References**


**The Debye Environment**

Alberto H. F. Laender is a professor of computer science at the Federal University of Minas Gerais, Brazil. His research interests include conceptual modeling, database design, database interfaces, Web data management, and semistructured data. He received a PhD in computing from the University of East Anglia, UK.

Altigran S. da Silva is an assistant professor of computer science at the University of Amazonas, Brazil. His research interests include conceptual modeling, database design, semistructured Web data, and Web information systems. He recently completed his PhD in computer science at the Federal University of Minas Gerais.

Paulo B. Golgher is a technology manager at Akwan Information Technologies, Brazil. His research interests include semistructured data, Web agents, and information retrieval. He received an MSc in computer science from the Federal University of Minas Gerais.

Berthier Ribeiro-Neto is an associate professor of computer science at the Federal University of Minas Gerais, Brazil. His research interests include information retrieval, Web information systems, digital libraries, and video on demand. He has a PhD in computer science from the University of California, Los Angeles.

Irina M. R. Evangelista Filha is a member of the database research group at the Federal University of Minas Gerais, Brazil, from which she received an MSc in computer science. Her research interest include databases, semistructured Web data, and query interfaces.

Karine V. Magalhães is a system analyst in the SIDS project at the Federal University of Minas Gerais, Brazil. Her research interests include databases, semistructured Web data, and software engineering. She received an MSc in computer science from the Federal University of Minas Gerais.

Readers can contact the authors at {laender, alti, berthier, imre, versieux}@dcc.ufmg.br or golgher@akwan.com.br.