Adapter Generation for Extracting and Querying Data from Web Sources

Kai-Uwe Sattler Michael Höding
Department of Computer Science, University of Magdeburg
P.O. Box 4120, D-39016 Magdeburg, Germany
{kus|hoeding}@iti.cs.uni-magdeburg.de

Abstract

Accessing and integrating data from heterogeneous sources has become a significant challenge. So-called *adapters* provide the functionality for translating SQL queries into queries understandable by the source as well as converting the results into a common model. In this paper, we present our approach of an adapter for Web sources, which is configurable by specifying a source-specific extraction function. We focus on two main tasks: query modification in order to extend the source capabilities and data extraction. The extraction step bases on an operational description, that enables an interactive exploration of the result format during the development phase. Finally, we present our ideas for semi-automatic discovery of extraction patterns by analyzing example documents.

1 Introduction

With the increasing popularity of the Web a wealth of information from many different domains has become available on-line. Besides semi-structured documents, numerous sources contain structured information, e.g. product information, stock exchange information, biographical or biological data. Therefore, the problem of integration has become a significant challenge. A number of research projects is devoted to this problem. The goal of projects like TSIMMIS [6], Information Manifold [10] or Garlic [5] is to provide an uniform and integrated access to different sources. Mediators [19] or federation services [16] play the central role in these integration systems and their main task is to collect and combine information from the sources. This involves translating user queries into source queries as well as extracting and merging the results. Typically, access to sources is provided by so-called *wrappers* or *adapters*, which convert data from the source into a common model and support a common query language.

Whereas the development of adapters for "real" databases may be simplified by using standard interfaces like ODBC or JDBC [8], the interface of Web sources is typically formed by HTML forms and CGI scripts as the calling interface and HTML as format of results. In addition, Web sources are mostly limited in their query answering capabilities [12]. Therefore, user queries have to be modified by the adapter in respect to the source capabilities. This involves processing of query operations in the adapter in addition to source queries.

Developing an adapter by hand is impractical for several reasons: the number of potential sources can be very large and the format of sources changes frequently [3]. However, only a small part of the adapter code depends on the source, the remaining parts are common among a wide range of sources.

Based on this fact and the analysis of other approaches [3, 9] for adapter generation we present our approach of configurable adapters. In this paper, we focus on the tasks of query modification and data extraction. After presenting the data model and the supported query language in section 2, we discuss in section 3 the modification step and in section 4 the process of data extraction. Section 5 describes the architecture of the adapter implementation and section 6 introduces our approach for deriving extraction patterns. Finally, we conclude the results.

2 Data Model and Query Language

For representing and integrating data from heterogeneous sources a number of models has been developed in recent years. Apart from relational and object-oriented data models some special adaptations for semi-structured data were proposed, e.g. OEM [15] from the TSIMMIS project. However, because a lot of Web sources base on relational data or data with fixed structure a object-relational model [17] might be a suitable solution. By using object-oriented features (e.g. inheritance) the subsequent integration may be simplified. Therefore, we have choosen a simple object-relational model based on the ideas presented in [13]. The model includes relations and classes. A relation contains tuples while a class contains objects. Attribute values of relations or classes may be atomic values, objects as well as sets of these. Both relations and classes are treated uniformly: every class is associated with a relation, where the set of tuples corresponds to the class extension. Consider the following example schema of an online book-shop:

```
type book (
    title varchar (20), authors set(varchar (20)),
    price float, isbn varchar (25),
    publisher varchar (30))
```

The choice of a query language is closely related to the data model. Beside standard query languages like SQL or OQL some languages for Web-based sources were proposed, e.g. Lorel for semi-structured data [1], W3QL [11] and WebSQL [14] for Web documents or various proposed XML query languages [7]. In our approach we have selected a SQL subset extended by the *in*-predicate for testing set memberships. An example query for the book-shop relation introduced above could be defined as follows:

```
select title, authors, price from book
where ('Saake, G.' in authors or
    'Heuer, A.' in authors) and price <= 100
```

By supporting an object-relational data model and a SQL-like query language we are able to support the integration of Web-based sources and (object-)relational databases as well as an uniform access via standard interfaces like JDBC.

3 Query Modification

Typically, Web sources support only a limited set of queries, for example a selection of only some attributes, only few comparison operators or conjunctive conditions. Therefore, queries have to be modified to meet the capabilities of the source. Possibly this involves reordering operations, performing operations by the adapter or decomposing the query into sub-queries and merging the results. The concerned operations include the well-known relational algebra operations: selection σ, projection π, union $\cup$, renaming β etc.

In order to translate a query into a form performable by a source we need a description of the source capabilities. In the following we assume a source containing a relation R_{src} and supporting a set of selection operations. A single selection is described as a conjunction $c = \bigwedge_{i=1}^{n} c_i$ of atomic conditions $c_i = a\theta v$, where a is an attribute of R_{src}, v is a value and $\theta = \{<, >, \leq, \geq, =, \neq, \in\}$ is an operator. We call the set of all supported selections $\Sigma_{R_{src}}$. A query, represented by the relational algebra expression q_s can be performed (that means, is directly or indirectly supported by the source), if each operations applied to the source relation is a supported selection of $\Sigma_{R_{src}}$: q_s is supported, if for each term $\omega(r)$ with $r = R_{src} : \omega \equiv \sigma_c \wedge c \in \Sigma_{R_{src}}$.

We have to transform a query q into a supported query q_s. Consequently not all possible queries can be transformed. At least one supported selection is required or the source has to enable selections with empty conditions. In the following, we consider only the transformation of selections, because projection, renaming and join are performed in the adapter on the results of the source operations. We decompose and reorder selections that each source operation is a supported operation of the source. In detail, query modification is performed by the following steps:

(1) Transform the selection conditions into a disjunction of atomic conditions or conjunctive combinations. This results in an expression: $c' = \bigvee_{i=1}^{n} c_i$

(2) For each term c_i determine a maximal subexpression $c_s \subseteq c_i$ where $c_s \in \Sigma_{R_{src}} \wedge c_s \neq \emptyset$

(3) Transform each term c_i representing a selection operation with $c_i \notin \Sigma_{R_{src}}$ into a nested selection $\sigma_{c_i \setminus c_s}(\sigma_{c_s}(R_{src}))$. The whole selection is now represented by the union of the transformed terms:

$$r = \bigcup_{i=1}^{n} \sigma_{c_i \setminus c_s}(\sigma_{c_s}(R_{src}))$$

(4) Finally, remove redundant selections on temporary relations by applying the transformation rule:

$$\sigma_{c_1}(R) \cup \sigma_{c_2}(R) \Leftrightarrow \sigma_{c1 \vee c2}(R) \text{ for } R \neq R_{src}$$

If successfully, this modification steps result in a query which is processed by performing one or more sub-queries. In the book-shop example we assume a source supporting selections where only the author and/or the title attribute may be queried. As a result, the query from section 2 is transformed into the following query expression:

$$\pi(\sigma_{\text{price} \leq 100}(\sigma_{\text{'Saake, G.' in } authors}(R_{src}) \cup \sigma_{\text{'Heuer, A.' in } authors}(R_{src})))$$

This query is answered by performing two source queries followed by a selection and projection on the union of both result sets.

4 Query Processing and Data Extraction

For query processing we have to distinct between local queries (performed by the adapter) and source queries evaluated in the source. Processing a source query involves two steps: (1) a HTTP request has to be sent to the source and (2) the received document has to be parsed in order to extract the result data. This requires an extraction function f_E, that returns the results for a given selection. While the step of translating the query into the HTTP request is straightforward, the extraction step needs more work.

In contrast to grammar-based approaches [3], where a parser is constructed from a given grammar for the resulting document, our extraction approach bases on an operational description for defining the extraction function. A function is composed from predefined utility functions for extracting substrings and creating result tuples. We have identified the following utility functions:

- get_url (url, params) $\longrightarrow$ *string*
  returns a string containing the document, which is addressed by *url* and *params*.

- extract (s, pattern) $\longrightarrow$ *tuple*
  extracts a tuple of strings from *s*, where *pattern* specifies a extraction template. This template is a regular expression containing HTML tags as special characters and placeholders $1 . . . $n for the tuple elements which have to be extracted.

- split (s, pattern) $\longrightarrow$ *list*
  splits a string *s* into a list of strings at the given delimiters.

- map (func, l) $\longrightarrow$ *list*
  applies a function *func* on every element of list *l* and returns the resulting list of tuples.

- tuple ($s_1, \ldots, s_n$) $\longrightarrow$ *tuple*
  constructs a tuple from a list of strings $s_1 \ \ldots \ s_n$.

With the help of these functions we are able to solve most extraction tasks. In addition, providing these functions as part of an interpreter language enables the interactive discovery of extraction steps for new sources.

Adopting this operational approach to our query adapter requires an execution unit as part of the adapter and a mapping between selections and the extraction functions.

5 Adapter Architecture and Implementation

In our adapter, only the extraction step depends on the Web source. Query modification and processing are similar for all sources. Therefore, we are able to simplify the adapter development by providing a framework containing the source-independent functionality, which may be configured by adding the extraction function.

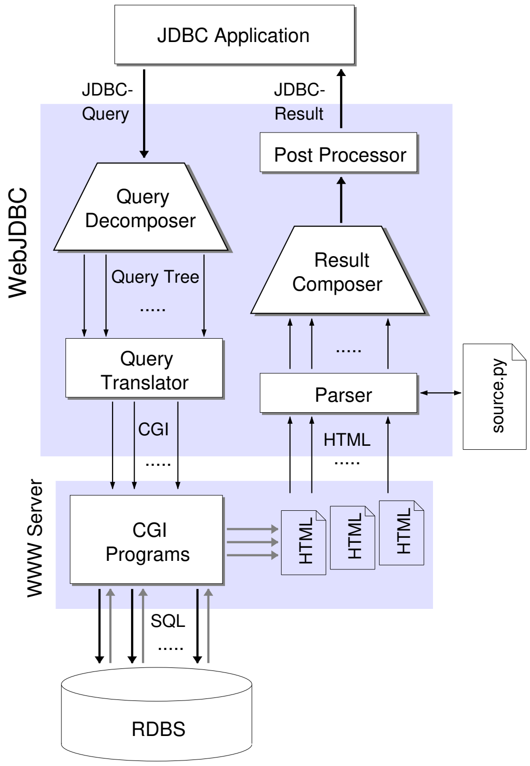

Figure 1: Adapter architecture

Figure 1 shows the architecture of the approach. A given user query is processed as follows. First, the query is decomposed according to the algorithm from section 3 and a query tree is constructed. Every leaf node in the tree corresponds to a source query, which is translated into a CGI request. For each resulting HTML document an extraction function has to be called by the parser. The extracted results of the sub-queries are composed and the remaining operations (e.g. selection, projection, union) are processed. Finally, the post processor performs necessary data conversations by using meta-data.

The presented approach was implemented as a JDBC driver. The driver supports SQL queries for Web sources, which provide at least a CGI interface. The execution environment for the extraction function is implemented by embedding a Python interpreter into the adapter. An extraction function is written as a Python [18] script, the utility function described in section 4 are implemented in Python, too. As an example, consider the following script for extracting results from the book-shop web site:

```
def extract_item (s):
    title, tmp, price = extract (s,
    '$1</a></b><dd>$2<br>$3<p>')
    authors = split (split(tmp, '/')[0], ',')
    price = match (price, '[0-9]+,[0-9][0-9]')
    return title, authors, price

def extract_books (params):
    doc = get_url (bookshop-url, params)
    items = split (extract (doc,
    '<html><body><dl>$1</dl>'), '<dt><b><a>')
    return map (extract_item, items)
```

In summary, configuring the adapter requires a schema definition, a description of selections supported by the source, the mapping between selections and the appropriated HTTP requests as well as Python script for extracting results. All this information is bundled in a configuration file, which is evaluated by the adapter at startup time.

6 Derivation of Extraction Patterns

Obviously the definition of extraction patterns might be one of the most expensive steps in adapter design. For the human designer it is often quite difficult to find suitable patterns for the extraction function by analyzing example documents [4]. One reason for that is the coexistence of useful data and a huge volume of other information (trash), e.g. headlines, banners for advertising, buttons, etc. Therefore, the designer needs supporting tools for pattern derivation [2]. For that we propose a semi-automatic approach, according to the following steps:

1. Define one or more example URLs, representing queries to the data source (interactive)
2. Load example Web pages (automatic)
3. Extract one or more example tuples from the example Web pages (interactive)
4. Find *prestrings*, *instrings*, *poststrings* for all example tuples in the example Web pages (automatic)
5. Evaluate and unify prestrings, instrings, poststrings; derive *prefix*, *infixes*, *postfix* (automatic)
6. Derive extract-functions for adapter configuration (automatic)
7. Solve multi-valued instrings, derive specific split-functions for adapter configuration (automatic)

Preconditions for the algorithm are the existence of constant prefixes and postfixes, which determine beginning and end of data areas (attribute values or tuples) and a suitable set of examples. The term *suitable* is quite fuzzy. For instance the example should contain neighbouring tuples. If the algorithm fails, the structure of tuples in the Web documents is not homogeneous and can not be parsed by our adapters. However, we have to point out, that Web pages generated by CGI programs accessing a database system generally support the necessary regular structure.

The following set of rules illustrates the main ideas of the extraction rule derivation[1]. The algorithm calculated a unique prefix and postfix for the tuple. Based on this it derives the following rule representing the structure of a result Web page containing a set of tuples.

```
tuple = split(extract(www_page,'tuple_prefix.$1'),
                    'tuple_postfix')
```

This rule defines the outer structure of a tuple. Defining the inner structure of the tuple one has to take care of set-valued attributes and not-constant instrings.

Beginning with flat tuples containing only atomic attributes which are separated by constant infixes, that are calculated by step 5 of the algorithm, the following extraction function is derived:

```
att1, att2, att3 = extract(s,'$1.infix1.$2.infix2.$3')
```

Obviously, separators are not always constant. Often separators, given by example tuples, contain additional but unused data. In this case the algorithm derives a constant left side and right side which function as postfix and prefix. In the following example the separator tmp between att1 and att2 is variable:

```
att1, tmp, att2, att3 =
    extract(s,'$1.postfix1.$2.prefix1.$3.infix2.$4')
```

[1] For easy read of the rules we separate postfix, placeholders, etc. by '.'. These dots are not part of the rules in the resulting source.py file.

The handling of set-valued attributes is more difficult. First there are two opportunities to define attribute values for such attributes in the example tuples. The first is to instantiate the value with the complete set of example attributes including separators. In that way the set-valued attribute can be handled in the same way as an atomic attribute. Beside this the attribute has to be defined as 'set-valued' manually. In an additional step the separator has to be computed. A disadvantage is, that in numerous cases the complete string representing the set in the example is really long. This can be caused by extensive use of HTML tags (especially links) or additional but unused information.

Therefore we discuss the use of only one example element of the set for each example tuple. (In the case of existence of set-valued first or last attributes, e.g. one author of a list of more authors, these approach influences the calculation of tuple prefixes or tuple postfixes.) For the calculation of prefix and postfix of the set and the separators the algorithm needs example elements from the beginning, the middle, and the end of a example set. Separators can be again constant and variable. In the following example att2 is set-valued and the set elements are separated by a (constant) infix:

```
att1, tmp, att3 = extract(s,'$1.infix1.$2.infix2.$3')
a_set = split(tmp,'$1,set_infix')
```

To support variable separators an additional extraction rule has to be derived as illustrated before.

The algorithm is implemented in Java and supports the semi-automatic and interactive derivation of extraction patterns.

7 Conclusion

Recently, there has been an increasing interest in integrating data from Web sources into databases. The required functionality is typically provided by adapters. In this paper, we have presented our approach of a configurable adapter. An adapter framework implements the source-independent tasks, like query modification and processing as well as data conversion. This framework ist instantiated for a specific source by defining an extraction script. An operational description of the extraction step enables the interactive exploration of the result document structure. Furthermore, we have discussed semi-automatic discovery of extraction patterns by analyzing example documents. A prototype of the adapter was implemented in Java as JDBC driver for querying Web sources with a SQL subset.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[2] A. Adelberg. NoDoSE - A Tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents. In L. Haas and A. Tiwary, editors, *SIGMOD'98, Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data, June 1–4, 1998, Seattle, Washington, USA*, volume 25 of *ACM SIGMOD Record*, pages 283–294. ACM Press, June 1998.

[3] N. Ashish and C. Knoblock. Wrapper Generation for Semi-structured Internet Sources. In *Workshop on Management of Semistructured Data*, Tucson, Arizona, 1997.

[4] S. Brin. Extracting patterns and relations form the world wide web. In Gianni Mecca, editor, *Proc. of WebDB98 - International Workshop on the Web and Databases*, pages 102–108, Valencia, Spain, March 1998. Springer.

[5] M. Carey, L. Haas, P. Schwarz, M. Arya, W. Cody, R. Fagin, A. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. Williams, and E. Wimmers. Towards Heterogeneous Multimedia Information Ssystems: the Garlic Approach. In *Proceedings of the 6th International Conference on Data Engineering*, pages 123–130, Los Angeles, CA, February 1995.

[6] 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 10th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

[7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. Submission to the World Wide Web Consortium, August 1998. http://www.w3.org/TR/NOTE-xml-ql/.

[8] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access with Java – A Tutorial and Annotated Reference*. Addison Wesley, Reading, MA, 1997.

[9] J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, and R. Yerneni. Template-Based Wrappers in the TSIMMIS System. In *Proceedings of 23rd ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[10] T. Kirk, A. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *Proceedings of the AAAI Spring Symposium Series*, March 1995.

[11] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World Wide Web. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 54–65, Zürich, Switzerland, September 1995.

[12] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22nd Interational Conference on Very Large Databases (VLDB'96)*, 1996.

[13] A. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems*, 5(2), September 1995.

[14] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. In *Proceedings of the Conference on Parallel and Distributed Information Syswtems (PDIS'96)*, December 1996.

[15] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across heterogeneous Information Sources. In *Proceedings of Data Engineering Conference*, Taipei, Taiwan, March 1995.

[16] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[17] M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, San Francisco, CA, 1996.

[18] A. Watters, G. van Rossum, and J. Ahlstrom. *Internet Programming with Python*. M&T Books, 1996.

[19] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.