Abstract

The world today is characterised by the proliferation of information sources available through media such as the WWW, databases, semi-structured files, etc. But, very often, this information is scattered, heterogeneous and weakly structured (semi-structured data). E-commerce is, by nature, one field in which this phenomena is specially strong. This paper overviews a mediator system for the construction of structured and semi-structured data integration applications. This system has been used in the construction of various applications on the Internet and in corporate environments, which are being used in real operational environments. Some conclusions arising from the experience gained from the use of such applications are also put forward.

1. Introduction

The world today is characterised by the abundance of information sources available through media such as the WWW, databases, semi-structured files, etc. But, very often, this information is scattered, heterogeneous and weakly structured (semi-structured data[2]). E-commerce applications are a specially important field where this kind of problems arise frequently.

Conventional approaches for data integration, such as data warehousing, present several drawbacks:

- They are not valid for building applications requiring up-to-date data
- They cannot handle semi-structured data sources, such as websites (HTML) or XML files.

Mediator systems, which were first proposed in [1], aim to solve these problems. In this approach, the data remains in the sources and the mediator is responsible for providing users with the “illusion” of being querying a single and coherent global schema. When the mediator receives a query, decomposes it into sub-queries on the sources and integrate the sub-results in order to obtain the query answer. Since data is retrieved from the sources at query time, it is guaranteed to be fresh. This approach is often called as virtual (as opposed to the warehousing approach).

For dealing with semi-structured data, mediator systems provide the so-called wrapper generation techniques, which allow to obtain structured views over semi-structured sources such as websites.

This paper describes an industrial system following such architecture, which allows the fast yet powerful extraction and combination of information from various heterogeneous, structured or semi-structured sources, to create an unified global schema, for said information. This system has been used for the construction of various industrial data integration applications, both in the Internet and Intranet environments, integrating information from sources such as Web sites, databases, spreadsheets, structured files, etc. The different integration applications constructed with this system currently handle more than 700 different sources.

Section 2 provides a general view of the system. Section 3 provides some conclusions obtained from our experience in using the system in e-commerce applications. Section 4 considers related work.
2. System Overview

The main components of the system and their interrelationships are shown in Figure 1. The physical layer is comprised of wrappers for different data sources: relational databases, web sites, flat files, spread sheets, etc. Wrappers are semi-automatically generated by a tool.

The logical layer, the so called Aggregation Engine, is the core of the platform containing the Query/Plan Generator, the Optimizer and the Execution Engine. An administration tool is used to manage the data dictionary. A cache module stores materialised views, avoiding querying the sources when queries can be solved using the cache. The Planning tool permits scheduling periodical data prefetches. The Security SDK allows ciphering of data when required.

We can ascertain two phases in a mediator operation: the integration system creation phase and the execution phase. The creation phase is performed by the system administrator, and its objective is to define the structure of the relations which will be part of the integration system global schema. The execution phase is that of the normal operation of the system, in which queries expressed in a language similar to SQL concerning global relations are accepted, and then answered by extracting and combining source information. We explain these two phases in detail hereinafter.

2.1 Creation Phase

This phase consists of three basic stages: modelling of the source relations, the wrapper generation and the definition of the global schema relations.

2.1.1 Source Model. In our model each source exports a combination of relations, called base relations. Each base relation is composed of a combination of attributes, each one belonging to a particular data type. Besides of the usual atomic data-types (strings of characters, integers, money, dates, etc.) structs and arrays are also supported.

Each relation will represent its query capability through what we term as search-methods. Each search-method is comprised of three elements:

- Negative capabilities: indicate the restrictions which must be satisfied by the queries sent to the relation through this particular search-method.
- Positive capabilities: Indicate the query capabilities offered by the relation, once the negative capabilities has been satisfied.
- Output attributes: Set of attributes which appear in the response of the queries sent against the relation. This set is needed since some sources allow being queried by attributes not included in query answers.
Both negative and positive capabilities are represented through a set of 4-uples with the format: (attribute, operator, multiplicity, possible_values) where:

- **attribute** is an attribute of the relation.
- **operator** is an operator valid for the attribute data type.
- **multiplicity** indicates how many query conditions for the given pair attribute/operator can execute the relation in the same query. ‘+’ indicates a number of values above 0 but with no upper limit.
- **Possible_values** is the list of values by which the attribute can be consulted. If it contains the value ‘Any’ it means that the search range is not limited (in the range associated to the attribute data type).

**Example 1:** We consider the example of an on-line bookshop whose search form is shown in figure 2.

The form forces the user to specify a value for the **TITLE** attribute (i.e. it is a mandatory field) and optionally the user can specify a value for the **AUTHOR** attribute, for the **FORMAT** attribute (restricted to a list of possible values) and a range of values for the **PRICE** attribute.

Queries by title and author are searches by keyword (operator **Contains**). An arbitrary number of words can be used to fill in the boxes and the source will perform the logical **AND** operation between them. For instance, filling the title box with the words ‘java’ and ‘xml’ would return all the books containing both words in its title.

In addition to the former attributes, we assumed that the shop also returns a **PUBLISHER** attribute.

![Figure 2. Bookshop search form](image)

We model this source as the following relation:

\[ R = \{ \text{TITLE: String, AUTHOR: String, FORMAT: Enumerated (Hardcover, Paperback, eBooks), PUBLISHER: String, PRICE: Money} \} \]

Whose capabilities are modelled by the following search-method:

```plaintext
{ 
  NEGATIVE { 
   (TITLE,Contains,1,ANY) 
  } 
  POSITIVE{ 
   (TITLE,Contains,+,ANY) 
   (AUTHOR,Contains,+,ANY) 
   (FORMAT,=,1,{'Hardcover', 'Paperback', 'eBooks'}) 
   (PRICE,<=,1,ANY) 
   (PRICE,>=,1,ANY) 
  } 
  OUTPUT {TITLE, AUTHOR, FORMAT, PUBLISHER, PRICE} 
}
```

Not all the relations can be modelled with only one search-method. When a relation presents mutually excluding options, more than a search-method is needed. For instance, it could happen that the source, instead of forcing the user to specify a value for the **TITLE** attribute, only forced to specify a value either for the **TITLE** attribute or for the **AUTHOR** attribute. This could be represented by adding a second search-method identical to the former one but substituting its negative 4-uple into: (AUTHOR,Contains,1,ANY).

### 2.1.2 Wrapper Generation.

The following stage in the creation phase is the construction of wrappers. Each wrapper must provide access to the source in a way that, when faced by the mediator, it behaves according to the model shown in the previous section. In our system, the wrapper generation process for Web sources, XML sources, JDBC databases and structured or semi-structured text files is performed with the assistance of a semi-automatic generation tool which enables them to be created and maintained in a fast and simple way.

Due to its importance, we have paid a special attention to web sources. The following section addresses that particular problem.

### 2.1.3 Wrapper Generation for Web Sources.

In the case of Web sources, our tool uses the approach of providing a specification language to generate specialised grammars for sources[3]. In addition, it provides graphical tools to construct the wrapper specifications in an interactive way by highlighting relevant portions of the
pages. This enables the wrappers to be generated, to a large extent, by staff with no programming knowledge.

Access to and navigation of Web sources are carried out by using the Microsoft Internet Explorer (MSIE) browser. Each access to a source in our system is identical to the process carried out by a user who connects to this source using MSIE.

We have defined a language which let wrapper generators easily define navigation sequences using MSIE. Consider the following example in order to get a flavor of the language.

**Example 2:** Let us suppose we want to model a web bookshop (e.g. Barnes & Noble) as a relation R which attributes are TITLE, AUTHORS, PUBLISHER, PRICE and DISCOUNT. The “advanced search” form lets the user perform queries by filling an author field and/or a title field.

The following sequence of commands is able to perform a search in such a form.

```java
Navigate("http://shop.barnesandnoble.com/booksearch/search.asp");
FindFormByAction("http://shop.barnesandnoble.com/BookSearch/results.asp");
SetInputValue("TTL", @TITLE);
SetInputValue("ATH", @AUTHOR);
Submit();
```

The ‘Navigate’ command makes the browser navigate to the given URL. Its effects are equivalent to that of a human user typing the URL on his/her browser bar.

The ‘FindFormByAction’ command looks for the first HTML form in the page with the given ‘action’ attribute value.

Then, the ‘SetInputValue(fieldName, value)’ commands are used to assign values to the form fields. The ‘value’ parameter might be a string (closed by “”) or a variable (prefixed by ‘@’). Variables are replaced by its value at execution time. For instance, if the wrapper receives the query 'select * from R where title="Java Programming"', then it would replace @TITLE by “Java Programming”.

The command Submit(), submits the form and loads the result page. Note that the form is submitted by generating a ‘submit’ event over the object representing the form in the browser DOM tree. We do not construct any HTTP request; instead, we delegate this task to the browser. This way, we do not have to deal with common and usually very complex issues as hidden session numbers, Javascript forms, and so on. These are transparent to our system, exactly in the same way that they are transparent to a human user navigating with his/her browser.

Our language also includes commands to cross links, deal with frames and pop-up windows, etc. Besides, as we will see, these sequences can be easily generated “by means of examples” (i.e. simply navigating). This way, no programming skills are required to perform the process.

Once the wrapper navigation sequence has been constructed (and its mapping with base relation search methods realized), a specification must be written which states how to extract the tuples from the HTML pages returned by the source. Consider the following simple example.

**Example 3:** The following figure shows a fragment of a response from the e-shop, Barnes & Noble.

<table>
<thead>
<tr>
<th>Beginning Java 2 - JDK 1.3 Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Stock: Ships within 24 hours</td>
</tr>
<tr>
<td>Ivor Horton / Paperback / Wrox Press, Inc. / March 2000</td>
</tr>
<tr>
<td>Our Price: $39.99, You Save 20%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The Complete Java 2 Certification Study Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Stock: Ships within 24 hours</td>
</tr>
<tr>
<td>Simon Roberts, Philip Heller, Michael Ernest / Hardcover / Sybex, Incorporated / September 2000</td>
</tr>
<tr>
<td>Our Price: $39.99</td>
</tr>
</tbody>
</table>

Let us suppose again we want to model this source as a relation R which attributes are: TITLE, AUTHORS, PUBLISHER, PRICE and DISCOUNT. A valid specification would be:

```java
PATTERN R {
    :TITLE ENDOFLINE
    IRRELEVANT ENDOFLINE
    :AUTHORS "/" IRRELEVANT "/"
    PUBLISHER "/" IRRELEVANT ENDOFLINE
    "Our Price:" :PRICE ", You Save" :DISCOUNT ENDOFLINE?
}
```

The basic idea is to construct a pattern in which the system will search for matches in the whole document. A pattern will consist mainly of text attributes to extract (here TITLE, AUTHORS, PUBLISHER, PRICE and DISCOUNT), text portions not to be extracted (denoted as IRRELEVANT) and separators.
Separator between attributes can be text separators ("/", "Our Price: ", etc.) or tag-type separators. A tag-type separator (ENDOFLINE) represents a regular expression concerning document format information (commonly HTML tags).

The tag-type separators can be defined to suit and reused in different specifications. ENDOFLINE, for example, could be defined through a regular expression of the style ("<br>"|"</p>"|"</tr>"|"</td></tr>"), and could be used in any specification for HTML sources. Normally this kind of separators are defined by a user who is something more of an expert and grouped into reusable sets. Then, these sets are used by less qualified wrapper generators to build pattern specifications for many sources.

Our experience shows that most applications need a very reduced set of tag-type separators. For instance, we have build a comparison shopping application which compares prices across more than 200 web sources using only one set, composed of the elements ENDOFLINE, TABULATOR (which represents a "<td>" tag) and ANCHOR (which represents an anchor HTML tag). Nevertheless, the ability to define any tag-type separator needed guarantees the system will be able to deal with any situation.

It is also possible to delimit optional portions (which are closed by the symbols '?') and to specify alternative portions (with the character '|').

To avoid pattern ambiguity, several constructions are provided to limit the search space for a pattern, identifying relevant regions in the page. The basic constructions are FROM and UNTIL which identify start and end patterns to a relevant region.

The system also allows for the easy handling of nested subpatterns inside the page (which are mappable to array and struct-type attributes of a relation). Another feature lets the user include navigation sequences inside the pattern in order to handle situations as tuples whose attributes are distributed in different pages or multi-page search results.

Our system also provides a graphical tool to ease the generation of navigation sequences and pattern specifications. An overview of its basic functioning follows.

The tool is basically an Internet browser (Microsoft Internet Explorer) which has been extended with several menus and buttons.

User can select the 'Begin sequence' option in order to make the tool change to "record mode", in which the tool records user navigation steps. Besides, it asks the user for additional information when it is required to complete a navigation sequence command. For instance, when the user navigates through a form, the system opens a pop-up window where the user is asked to assign values to the form fields. Constant values or wrapper-level meaningful variables (as @TITLE and @AUTHOR in the previous example, telling the wrapper "what these form fields represent") can be assigned.

When the user arrives at a page where he or she wants to extract data from, he/she highlights one or several relevant regions of the page. Then, he or she chooses in the page one occurrence of the tuples to be extracted, and selects the option 'Generate Pattern'. Then, the tool asks the user for the name of the pattern he wants to generate. This is important because it is possible to assign some default metainformation to a pattern name in order to improve the system "guesses". This metainformation will include at least a set of tag-type separators to use, and might include additional information such as common text separators in the domain and examples of values for the pattern attributes. For instance, the following simple pattern metainformation could be used to a 'Book' pattern in a comparative shopping application:

```
ATTRIBUTES={TITLE, AUTHORS, PUBLISHER, PRICE, DISCOUNT}
TAG-TYPE_SET={ENDOFLINE}
TEXT_SEPARATORS={"/", ";")
EXAMPLES
{
    TITLE=@TITLE
    AUTHORS=@AUTHORS
}
```

The '@TITLE' and '@AUTHORS' values assigned to the TITLE and AUTHORS attributes, are used to inform the system that it should use as examples the values provided by the user for these attributes when he or she filled the last HTML form.

Then, the tool obtains the HTML code associated to the highlighted tuple and gives it to the parser. The parser first divides it according to the tag-type separators set, obtaining a preliminary pattern. Then it tries to divide the texts obtained according to the common text separators indicated in the pattern metainformation (if they exist). Finally, it attempts to assign each text portion to an attribute of the pattern by comparing them with the available examples. If it is not possible for a given text portion, it is assigned the name 'IRRELEVANT'.

For instance, if the user highlighted the first Barnes & Noble book search result showed in the former example, the generated pattern would be:
Now the user can correct the pattern by: 1) Assigning the right attribute name to the attributes misidentified by the system (e.g. Assign the attribute ‘PUBLISHER’ to the third text token in line three) 2) Divide or concatenate text tokens adding or removing text separators. Simple graphical wizards are provided to guide the user in these tasks.

When the pattern is complete, it can be tested by trying to extract all the relevant tuples from example pages. If the pattern does not extract all the required tuples, then the user can add a new pattern by highlighting one of the missing tuples, thus making the system to also match the instances following that format. If the pattern matches more tuples than the required ones (i.e. the pattern is ambiguous), the user can delimit a search-region for the pattern using the mouse. This iterative process ends when exactly the required instances are matched.

When the pattern is already tested, the user can also choose to use it and obtain new examples for an attribute of the pattern. For instance, ‘Wrox Press’ and ‘Sybex Incorporated’ could be new examples for the PUBLISHER attribute.

2.1.4 Definition of the Relations of the Global Schema. Once the base relations have been defined and their wrappers constructed, each relation of the global schema is defined by a query concerning the base relations, in a similar way to the definition of views in a conventional database. This approach is known in mediator literature as Global As View\[3\]. The query is expressed in the same language in which the queries in the execution phase will be written and can include selections, projections, unions, joins, orderings and aggregations. It should also be pointed out that a view can also be defined from previously defined intermediate views.

Example 4: Consider the three base relations:

A={TITLE:String, AUTHOR:String, PRICE:Money}  
B={TITLE:String, AUTHOR:String, PRICE:Money}  
C={TITLE:String, AUTHOR:String, RELEVANCE:Integer}

A and B represent two Internet book merchant sites. C represents a web source in which its users assess the books’ relevance, and the system enables searches for the average relevance of a certain book. Consider that we want to obtain a global schema relation:

\[R=(\text{TITLE, AUTHOR, PRICE, RELEVANCE})\]

which contains all the books from A and B (only one tuple for each book), together with their average relevance according to C, and the minimum price for each book found in the two e-shops. The definition of R in our system would be:

\[
\text{SELECT TITLE, AUTHOR, RELEVANCE, MIN(PRICE)}  
\text{FROM (I SELECT TITLE, AUTHOR, PRICE FROM A}  
\text{UNION SELECT TITLE, AUTHOR, PRICE FROM B), C}  
\text{WHERE I.TITLE=C.TITLE AND I.AUTHOR=C.AUTHOR}  
\text{GROUPBY TITLE, AUTHOR, RELEVANCE}
\]

As has already been mentioned, the base relations can be limited in terms of their query capabilities. When the global schema relations are defined, the mediator is capable of “propagating” the query capability through the view tree, so that the capability allowable by the relations of the global schema are automatically obtained, in the form of a list of search methods with the same format seen in the search capability definition. This allows the mediator to know in advance if a certain query is going to be answered and also makes it possible for a mediator to be a source for other mediators. The query capability propagation mechanism from the base relations to the global schema relations is relatively complex and is not dealt with here. See [16] for a complete description.

2.2 Execution Phase

Once the creation phase has been completed, the mediator can accept SQL queries. When a query is received, the query interpreter translates it to the mediator’s internal format, which is an extension of relational algebra with some additional capabilities. Then it checks whether the query is answerable according to the global relation capabilities. If it is not, it rejects the query. If it is, the process continues.

The first step in executing the query is to reformulate it according to the sources. The mediator’s plan generator then creates all the possible execution plans for the query. The plans will normally differ from each other in aspects
such as the method used to execute the joins (conventional join, nested join, etc.) or the specific search methods to be used on the sources. It is the responsibility of the optimiser module to obtain the cost of each one of the plans, so that the best one can be selected. To evaluate its operation, the optimiser uses statistics, collated by the wrappers, regarding query costs in the base relations. Using this information, the optimiser module can travel up to the top of the tree for each query plan to obtain its cost.

Once the optimum plan has been chosen, the execution engine becomes responsible for executing it. Broadly speaking, it travels the chosen plan tree from the roots up to the leaves to achieve this. Given the tree’s construction, each leaf can be seen as a representation of a sub-query which involves only one source. Each one of these sub-queries is sent to the source wrapper and executed by it. Access to the sources is made in parallel.

The execution engine then combines the results returned by the sources in accordance with what is specified by the plan tree, obtaining in such a way the final response to the query. The system functions asynchronously. What this means is that as the results of the sources become available, the system start to process them, although the sources have not yet delivered a complete result.

Another important aspect is that, optionally, some or all of the relations could have been configured to keep a cache of source data (i.e. materialised views). A scheduling tool is also provided in order to execute periodical pre-fetches of selected data.

3. Experience obtained in using the system

This system (in successive versions) has been used to construct various commercial applications which are currently being used in real operational environments.

Applications constructed up to now can be classified in two groups:
- Search, comparison and aggregation applications on the Internet.
- E-business data integration applications.

In the first group the following, among others, have been constructed: comparison shopping applications, job offer searches, metasearch of Web sites, news searches in on-line press, financial aggregation, etc. In these cases, all the sources on which the mediator operates are Web sites. This combination of applications now accesses a total of over 500 different sources, and they are in operation in various Spanish Internet portals.

In this group of applications, the methods by which source data is combined tend to be simple (mainly unions, selections and projections), and the greatest difficulties are encountered in the construction and maintenance of wrappers. The reason is that we come across a high number of sources and that these also present a degree of complete autonomy. At the moment, average wrapper regeneration times in our system for each one of the main application domains varies between an average of 18.2 minutes for comparative purchase and metasearch sources, and an average of 43 minutes for financial aggregation sources.

In the second group, typical application scenarios include CRM (Customer Relation Management), where the customer data is distributed across many heterogeneous data repositories, EIPs (Enterprise Information Portals), where our system is used to provide an unified view over the heterogeneous content to be delivered through the portal and EAI (Enterprise Application Integration), where our system is used to unify data repositories in order to make easier the application integration at a higher level.

In these cases, the mediator normally operates on a mix of Web sources, relational databases, spreadsheet systems and semi-structured file systems (e.g. XML documents). In addition, the data is normally combined in a much more complex way than in the applications in the previous group, although there do tend to be fewer sources. The problem of wrapper maintenance hardly arises in these scenarios as there is a smaller number of sources, they change less frequently, and when they do change the administrators can be notified beforehand.

Our experience clearly shows that mediator-based integration solutions are much cheaper, faster and far less intrusive than other alternatives. An approach based on the construction of a new repository containing and unifying all the data to be integrated leads to a lengthy and costly process, which also involves redoing the applications which used the data in their old formats and retrain the users of these applications. On the other hand, the approach whereby a data warehouse is constructed, although not requiring the rewriting of the old applications, is still considerably more expensive and it is not valid for applications requiring up to date data.

4. Discussion and Related Work

In recent years, a significant amount of academic mediator systems has been developed, such as Tsimmis [4], Ariadne [5] or Hermes [6]. Various specific aspects in the construction of mediator systems have also been studied by the research community: wrapper generation for Web sources [7][8][9], query optimisation [6],
reformulation mechanisms [10], calculation of mediator capability [11], etc. A comprehensive survey can be found in [3]. These research systems do not deal with all the complexity one can encounter in real scenarios: performance, need of a flexible schema concerning materialization of views (virtual vs. warehouse), computing query capabilities or wrapper maintenance for existing highly complex commercial web sites.

In industry, much attention has been paid to mediator systems in recent times. Application specific systems have been developed in domains such as financial aggregation or comparison shopping (Yodlee [13], Teknowledge [14], MySimon [15]). General-purpose systems such as Nimble Integration Engine [12] from Nimble Corporation also have appeared in the market. This system follows a similar mediator architecture, however relies on XML data sources, and it provides a X-Query interface, instead of the more extended SQL interface supported by our system.

Some distinctive features from our system follows:

1) Automatic calculation of the query capabilities of the global schema relations using a simple but powerful capability description mechanism. Our experience says this is a “must-have” feature since it makes possible to use mediators as source for new ones, easily enabling incremental data integration processes and it enables users to know in advance the queries supported by the mediator. As far as we are aware, this problem was addressed only in [10], but using a capability description framework not expressive enough. Our approach to this problem is detailed in [16].

2) Powerful wrapper generation tools to allow reduced maintenance cost for existing highly complex commercial web sites. Our experience suggests that most of the proposed approaches will fail in effectiveness when dealing with this kind of sources, mainly because they forget about some of the most significant problems encountered when wrapping real web sources. Not only parsing is needed: translating a query into a navigation sequence, and then performing that sequence, is often at least as complex. Our language for navigation-sequences definition makes these complexities transparent to wrapper-generators.

3) SQL based engine. This makes integration in industrial environments much easier.

5. References