Structures in the Web*

Paolo Atzeni[1], Alessandro Masci[1], Giansalvatore Mecca[2], Paolo Merialdo[1] and Elena Tabet[1]

[1] Dipartimento di Informatica e Automazione
Università di Roma Tre – 00146 Roma
tel: 06/55177053, fax: 06/5579295
{atzeni, masci, merialdo, tabet}@inf.uniroma3.it

[2] D.I.F.A. – Università della Basilicata
via della Tecnica, 3 – 85100 Potenza
tel: 0971/474 638, fax: 0971/56537
mecca@dis.uniroma1.it

Abstract. The paper discusses the issue of views in the Web context. We introduce a set of tools and languages for managing and restructuring data coming from the World Wide Web. We present a specific data model, called the ARANEUS Data Model, inspired to the structures typically present in Web sites. The model allows us to describe the scheme of a Web hypertext, in the spirit of databases. Based on the data model, we develop the ARANEUS View language, to support a sophisticate view definition process; the language has two main components: the first, called ULIXES, is used to build database views of the Web, which can then be analyzed and integrated using database techniques; the second, called PENELOPE, allows the definition of derived Web hypertexts from relational views. This can be used to generate hypertextual views over the Web.

1 Introduction

Database Systems offer efficient and reliable technology to query structured data. However, because of the explosion of the World Wide Web [13], an increasing amount of information is stored in repositories organized according to less rigid structures, usually as hypertextual documents, and data access is based on browsing and information retrieval techniques.

Since browsing and search engines present severe limitations, several query languages [24, 26, 28] for the Web have been recently proposed. These approaches are mainly based on a loose notion of structure, and tend to see the Web as a huge collection of unstructured objects, organized as a graph. Clearly, traditional database techniques are of little use in this field, and new techniques need to be developed.

* This work was supported by Università di Roma Tre, MURST, and Consiglio Nazionale delle Ricerche.

In this paper, we explore an alternative approach; we aim at studying the applicability of more traditional database techniques to the Web framework . In order to do this, instead of considering Web data as essentially non-structured, we try to describe and exploit the structure according to which information is organized. Indeed, in many cases Web data are quite structured, that is, pages are organized in a rigid way, and present strong regularities. This is due to the fact that many Web sites are essentially interfaces towards traditional database systems, i.e. actual data are kept in a DBMS and pages are automatically produced starting from the database;[1] clearly, in these cases, the hypertext organization somehow reflects the structure of the underlying database. This is especially true in *Intranet* applications, where the Web essentially acts as a corporate information system providing access to databases, designed and mantained by administrators. On the other side, webmasters of large sites often need to adopt *ad hoc design methodologies* [18, 21], based on precise data models, in order to reduce design, data management, and maintenance costs. Because of these reasons, we can consider these sites as highly *structured.* It is important to note that this structure tends to be stable, with rare modifications; in fact, it is difficult to reorganize a huge amount of data, and users do not like to browse information whose organization often changes.

The Araneus project aims at defining methods and tools for the management of Web sites. In this paper we discuss some of these tools, and concentrate on the activity of querying a site. We first present a simple, *page oriented* data model, called the Araneus *Data Model* (adm), to be used to describe the structure of sites of interest. Then, based on the scheme, we show how it is possible to define *views* over the site using the *Araneus View Language* (avl). The view language has two main components: the first one, called Ulixes allows to define *relational* views over Web data; these views can then be queried using any relational query language. The second one, called Penelope, allows the definition of new structures from relational views, thus allowing the user to restructure query results and access them as an hypertext or to define *derived* Web sites.

Experiencing our approach on existing sites taught us two important lessons: first, from the logical viewpoint, the ways Web data are structured are often very similar to the ones common in database systems, although the physical organization may be different; moreover, the knowledge about the structure highly facilitates the process of querying the site, increasing the flexibility and effectiveness of query languages. We have modeled several existing Web sites from different domains, ranging from book publishers to university sites and virtual museums: adm allows to describe the site structure in a natural and intuitive way, it is easy to use and its constructs reflect the user's perception of the site. The resulting schemes are concise and describe the content of the sites in a synthetic and complete fashion. On these sites, the view definition language can be effectively used to access data, possibly correlating pieces of information from different sites. In this sense, the main contributions of the paper consist in the data model and the view definition language.

[1] Many commercial database systems now offer these functionalities.

The paper is organized as follows. Section 2 contains an overview of Araneus; Section 3 presents the Araneus data model; Sections 4 and 5 develop the view definition languages. Finally, the architecture of the Araneus system and some related work are discussed in Section 6. Many examples in the paper refer to pages in the Web site of the Uffizi Museum in Florence [3].

A prototype of the Araneus system is currently under development at the Università di Roma Tre. A demo of the system can be found at [1].

2 Overview

The Web is usually considered as a collection of data with little structure and high dynamics. In fact, other query languages proposed for the Web (see, for example [24, 26, 28] or, with a different perspective, [29, 7]) make essentially no assumption on the structure of data.

In this paper, we undertake a different approach, aimed at investigating the benefits of structure in querying Web data; differently from other approaches, we exploit the presence of structure to study the applicability of traditional database techniques to the Web context. We show how, in many cases, the management of Web data can highly profit from database technology. In fact, Web data often have structure. There are two main sources of structure in the Web. On the one hand, there is the *hypertextual* structure, according to which pieces of information logically connected are usually physically linked too. On the other hand there is the *textual* organization, due to the nature of HTML [19] documents. Data is organized inside text in many different ways, using the primitives of the HTML language. For example, given an HTML page, it is easy to recognize its title, or also some complex structures, like for example, lists of items, based on HTML *tags*.

In many cases, these structures are rather tight, so that we can assimilate the site, from the logical viewpoint, to a conventional database, in the sense that pages can be reconducted to a relatively small number of different types; this requires that pages referring to similar concepts have the same structure, that is, can be considered as having the same "scheme" and the number of different schemes is much smaller than the number of pages in the site. There are many examples of this kind: many university sites can be considered on-line databases about courses, instructors, departments, and research activities; similarly, some computer manufacturers offer highly structured information about products and prices; another example, the one we shall use throughout the paper, is that of *on-line museums*, in which pieces of art, artists, and rooms can be visited.

It is worth noting that this approach generalizes the models adopted in other query languages for the Web [24, 26, 28], in which pages are considered as essentially unstructured objects; this means that ADM can be used also when there is little structure, as shown in [12]. However, in addition, it allows to model existing structures and regularities, when considered interesting. In fact, in those sites in which pages referring to similar concepts can be considered as having the same scheme, the model provides a concise and effective description of the site's content.

We have developed several tools and methods for managing data coming from the Web. In this paper we mainly focus on the querying process; however, many of the ideas developed in the paper can be applied to the *design* process of a Web site as well. These latter aspects will be dealt with in a forthcoming paper [11].

Our approach to querying the Web consists in deriving the *scheme* according to which data are organized in the site, and then use this scheme to pose queries in a high level query language. To describe the scheme, we use a new data model, called the Araneus *Data Model* (ADM). We say that ADM is a *page oriented* model, in the sense that the main construct of the model is that of *page scheme*. Each page scheme describes the structure of a set of homogeneous pages in the site; the main intuition behind the model is that HTML pages can be seen as objects with an identifier, the URL, and several attributes, one for each relevant piece of information in the page. The values of these attributes can be usually extracted by accessing the HTML source code and applying suitable *text restructuring* procedures. The attributes in a page can be either *simple*, like text, images[2] or links to other pages, or *complex*. Complex attributes are essentially lists of items, possibly nested. Based on this perspective, the scheme of a Web site can be seen as a collection of *page schemes*, connected using links. This scheme is essentially a *structured view* of the site: on such view we can then pose queries using some high level query language.

To provide a flexible paradigm to access data, we reconsider the issue of path expressions in this framework. We introduce the notion of *navigational expression* as a means to express a set of navigations in the site; in fact, to access data in the site, it is natural to start from some *entry-point*, like, for example, the home page, and navigate until data of interest are found. Based on this notion of navigation, we define a flexible *view definition language*, called the Araneus *view language* (AVL). The view language has two components, to supports a two-way data-restructuring process. The first component, called Ulixes, is used to build *relational views* over Web data. The main idea, here, is that each navigation can be seen as a tuple of values and each navigational expression as a relation. These relations can be locally materialized and queried using *any* relational query language. Besides, considering the hypertextual nature of Web data, we often want to see query results not as a table, but as a *derived hypertext*, that is, a local set of pages containing data returned by the query in hypertextual form; this hypertext, which is again a *view* over the original site, can then be explored using a Web browser. The second component, called Penelope, supports this process: it allows to define new page schemes and present tuples in a relational table under the form of derived pages. Based on these ideas, a possible querying process can be summarized as follows: *(i)* the user first looks at the site scheme and identifies data of interest; *(ii)* the view language is then used to define a set

[2] It is possible to include other multimedia types as sound, postscript documents, movies and so on. Here, for the sake of simplicity, we do not consider these aspects.

of relational views over the site, which in turn can be locally queried using any relational query language; *(iii)* finally, table-based query results are restructured and local pages are created and returned to the user.

3 The Araneus Data Model

The Araneus Data Model (adm) is used to describe data contained in Web sites. We say that it is *page oriented* in the sense that it recognizes the central role that pages play in this framework. Each Web page is considered as an object with an identifier (the URL) and a set of attributes. Since we want to model the common features of similar pages, we introduce the notion of a *page scheme*, which resembles the notion of relation scheme in relational databases or class in object-oriented databases. Attributes of a page may have simple or complex type. Simple type attributes are *monovalued* and correspond essentially to text, images or *links* to other pages.[3] Consider for example the pages describing paintings at the *Uffizi virtual museum*;[4]. We can see that there is a set of elements that appear in each of these pages, such as a small image of the painting (which is also a link to a different page containing a larger image), the name of the painter, the title, and so on. It is natural to model the structure of these pages as a page scheme with several attributes, which include `MiniImage`, of type `IMAGE`, with the associated link, `ToImage`, to a larger image; `Painter`; and `Title` both of type `TEXT`.

Beside monovalued attributes, Web pages often contain (ordered) collections of objects, that is, *multivalued* attributes. We model them using *lists* of tuples. Consider, for example, the page in the Uffizi site listing all paintings in room number 9 of the Gallery[5]. It has some simple attributes again, such as room number and room name. However, it also has a *complex* attribute, that is, the list of paintings in the room. For each painting, the painter, the title and an optional link to the painting page are included. This is essentially a multivalued attribute of the page and can be naturally modeled as a list, each element of which contains information about one painting, namely the `Painter`, the `Title`, plus an optional link to the corresponding painting page. Component types in lists can be in turn multivalued, and therefore nested lists may arise. It should be noted that we have chosen lists as the only multivalued type since repeated patterns in Web pages are physically ordered.

There is one specific aspect in this framework with no counterpart in traditional data models: usually a site includes pages that have a special role and are

[3] Strictly speaking, an HTML *link* is a pair *(anchor, reference)*, where the *anchor* is of type text or image, and the *reference* is the URL of the destination page. However, to simplify the formalism, we ignore anchors, assuming that, if needed, they are modeled as independent attributes.

[4] See, for example, the *Birth of Venus* by Botticelli at `http://www.uffizi.firenze.it/Dipinti/botveneE10.html` and the *Sacrifice of Isaac* by Caravaggio, at `http://www.uffizi.firenze.it/Dipinti/caraisaccE16.html`

[5] `http://www.uffizi.firenze.it/Sale/salaE9.html`

"unique," in the sense there there are no other pages with the same structure. Typically, at least the home page of each site falls in this category. For the sake of homogeneity, we also model these pages by means of page schemes. See, for example, the page containing the list of all paintings having works in the Uffizi museum[6]; the page contains a nested list of elements, one for each painter; the painter's name, some biographical information, a list of rooms (those that contain his paintings), and a list of paintings are reported. This page is clearly *unique*, in the sense that no other page of the web site has a similar structure.

In order to formalize, as usual in models with object identifiers, we need two interrelated definitions, for types and page schemes, as follows.

Definition 1. **[Types]** Given a set of *base types* containing the types *text* and *image*, a set of *attribute names* (or simply *attributes*), and a set of *page scheme names*, the set of ADM *types* is recursively defined as follows (each type is either monovalued or multivalued):

- each base type is a *monovalued* ADM *type*;
- LINK TO P is a *monovalued* ADM *type*, for each page name;
- LIST OF$(A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n)$ is a *multivalued* ADM *type*, if A_1, A_2, $\ldots$, A_n are attributes and $T_1, T_2, \ldots, T_n$ are ADM types; attributes may be labeled as *optional*;
- nothing else is an ADM type.

Definition 2. **[Page scheme]** An ADM *page scheme* has form $P(A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n)$, where P is a page name, each A_i is an attribute and each T_i is an ADM type. Attributes may be labeled *optional.* The page scheme may be labeled as *unique.*

The page schemes discussed in the previous examples can be defined as follows using the ARANEUS *data definition language (DDL)*, whose syntax naturally follows from the previous definitions. Painting pages have the following scheme, in which only monovalued attributes appear:

```
PAGE SCHEME PaintingPage:
             Painter:     TEXT;
             Title:       TEXT;
             Date:        TEXT;
             Description: TEXT;
             MiniImage:   IMAGE;
             ToImage:     LINK TO ImagePage;
END PAGE SCHEME
```

Pages for rooms have a multivalued attribute, i.e. the list of paintings in the room, and can be described as follows:

[6] http://www.uffizi.firenze.it/indiceartisti.html

```
PAGE SCHEME RoomPage:
            RoomNo:     TEXT;
            RoomName:   TEXT;
            PaintList:  LIST OF (Painter:     TEXT;
                                 Title:       TEXT;
                                 ToPaint:     LINK TO PaintingPage
                                                     OPTIONAL);
END PAGE SCHEME
```

Note that pages may have a nested structure, like the page containing the list of all artists:

Now, we define the notion of ADM *scheme* simply as a set of page schemes.

Definition 3. **[Scheme]** Given a set of page names, $\mathcal{P}$, an ADM *scheme* is a set of page schemes, exactly one for each of the page names in $\mathcal{P}$.

We can represent the scheme as a directed multigraph; nodes in the scheme graph are page schemes; each unique page scheme is denoted as a single page, whereas non-unique page schemes are represented as to "stacks" of pages; edges are used to denote links. A fragment of the Uffizi Web site scheme is shown in Figure 1, which also contains an explanation of the other graphical primitives. Similarly, Figure 2 represents a fragment of the Louvre Web site [2].

Note that, to see actual pages in the site as instances of the page-scheme, we need to access the HTML source and apply suitable text restructuring procedures. Attribute values are extracted from the HTML source using the EDITOR language [10], a formalism for text manipulation. In the current implementation we wrap pages using Java classes; every page-scheme in the site corresponds to a specific class, with one method for each attribute; each method implements an EDITOR program that accesses the HTML source and returns a complex value for the attribute.

Based on this perspective, at the instance level, a site can be seen as a graph in which links connect trees corresponding to pages. In fact, each instance of a page-scheme is a tree (because of its nested structure), and may contain links to other instances. Nodes of trees (and therefore of the overall graph) are essentially tuples; each tuple attribute may either have a simple value or be the root of a subtree; optional attributes may have a null value.

4 ULIXES: Accessing Data in a Web Site

The presence of a structure in the site, described by means of a scheme, allows us to express *queries* on the Web. The query process is based on the notion of *navigation* in the site graph. In fact, note that a site offers in essence a set of *navigations*, i.e. paths in the site graph; these navigations allow to follow links between different pages, but also to explore the hierarchical structure of a page: they represent a natural means to query the page. Consider again the Uffizi site,

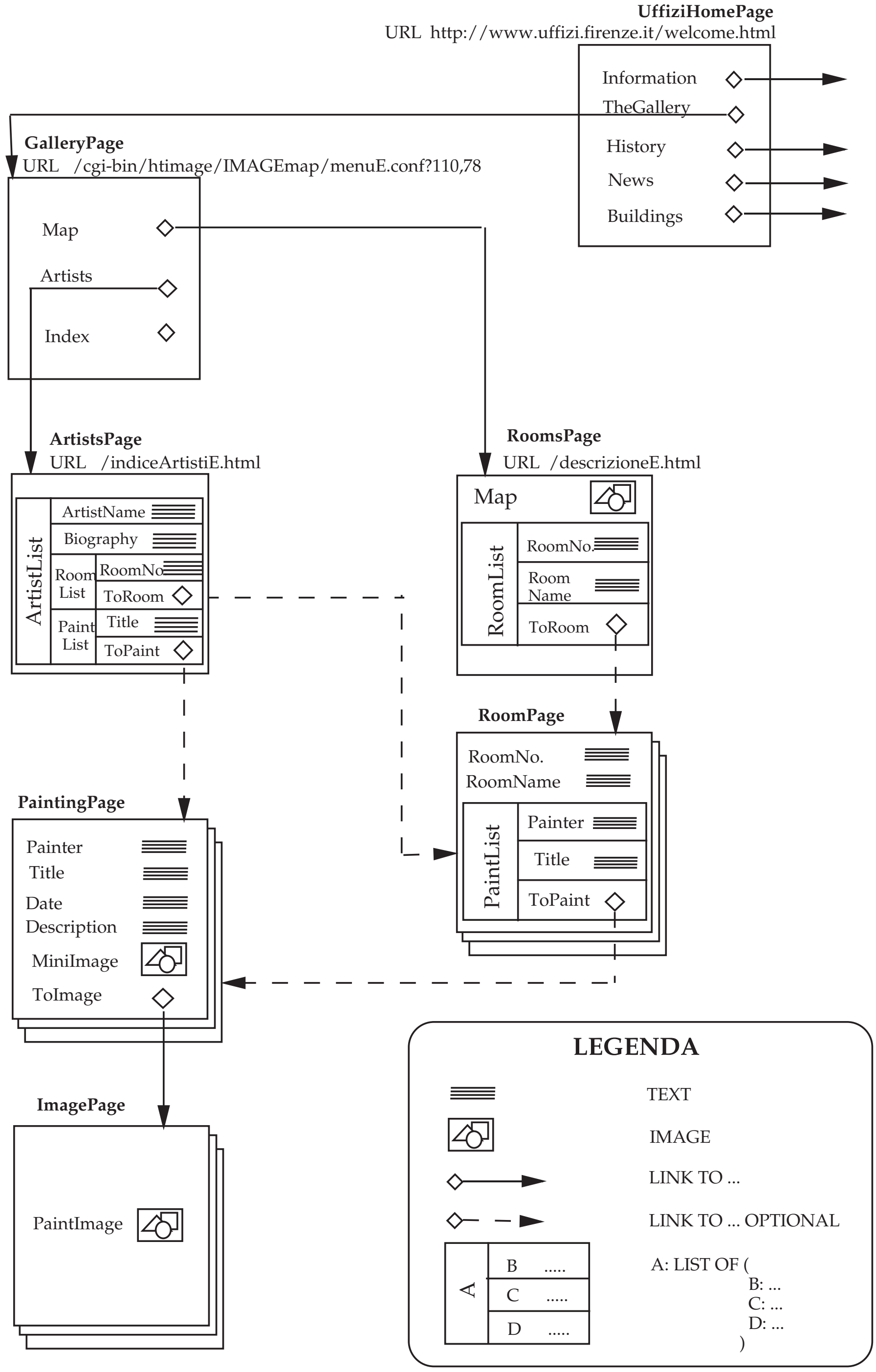

Fig. 1. A portion of `UffiziScheme`, the ADM scheme of the Uffizi site

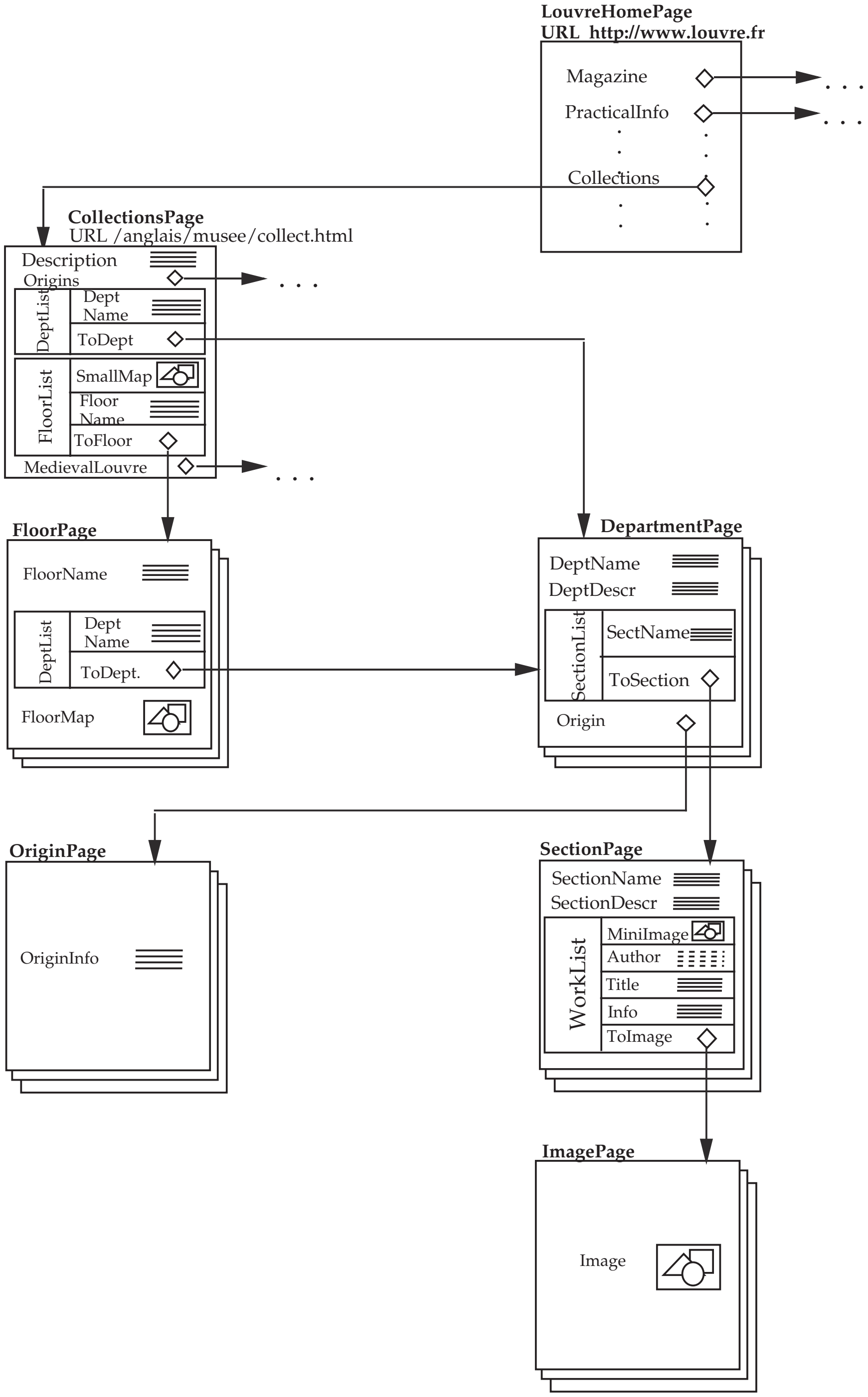

Fig. 2. A portion of `LouvreScheme`, the ADM scheme of the Louvre site

and suppose we are interested in the titles of all paintings in rooms of the virtual museum. Based on the scheme in Figure 1, we know that one way to answer the query is by navigating data in the following way: we start at the page containing the list of all rooms[7] and follow all possible paths in the page tree; some paths end with a link to a room page; we cross the link, and reach the corresponding room page; then we can navigate to reach all paintings titles. These navigations can be denoted with the following *navigational expression*, in which we use the dot operator (.) to navigate *inside* pages, and the *link operator* (→) to follow links:

$$\texttt{RoomsPage.RoomList.ToRoom} \rightarrow \texttt{RoomPage.PaintList} \qquad (1)$$

The semantics of the expression is easily interpreted as *all* possible *navigations*, i.e. paths, in the site graph starting with the unique page over page scheme `RoomsPage`, traversing each element in `RoomList`, following the associated link (if it exists), reaching a room page and ending with a painting in `PaintingList`.

It is worth noting that each of these navigations can be represented as a *tuple* of values, one value for each *monovalued* attribute associated to nodes in the navigation; thus, each navigational expression can be represented as a *relation*, in the relational model sense.[8]

To give another example, suppose now we are interested in all data about on-line paintings in the virtual museum. These can be reached using the following navigational expression:

$$\texttt{RoomsPage.RoomList.ToRoom} \rightarrow \texttt{RoomPage.PaintList.ToPaint} \rightarrow \\ \texttt{PaintingPage}$$

corresponding to all possible navigations in the site graph from `RoomsPage` to painting pages. In this case, only paintings for which a page exists are considered. Also in this case, a relation can be associated to this expression.

It is now straightforward to give the semantics of a navigational expression as a set of navigations, i.e. paths in the site graph. Given a navigational expression, N, we call $\textsc{sem}(N)$ the corresponding relation. We assume that attributes are suitably renamed whenever needed.

Given the relational nature of navigations, the definition of relational views over ADM schemes can be directly based on navigation specifications. We have a `DEFINE TABLE` statement to be used for this purpose, with the form:

```
DEFINE TABLE  R(B_1, B_2, ..., B_n)
AS            N
IN            S
USING         A_1, A_2, ..., A_n
```

[7] http://www.uffizi.firenze.it/descrizioneE.html

[8] In fact, a navigational expression could be seen as a list, that is, an ordered (multi)set of tuples. However, in this context, ordering is irrelevant and duplicates meaningless.

where: *(i)* R is a relation name and $B_1, B_2, \ldots, B_n$ are attributes; *(ii)* S is an ADM scheme; *(iii)* N is a navigational expression over S; and, *(iv)* $A_1, A_2, \ldots, A_n$ are attributes of $\text{SEM}(N)$. The semantics of a `DEFINE TABLE` can be immediately defined on the basis of previous notions: relation R is defined as the projection of $\text{SEM}(N)$ onto $A_1, A_2, \ldots, A_n$, with each A_i renamed to B_i, that is:

$$R = \rho_{B_1 \leftarrow A_1, \ldots, B_n \leftarrow A_n}(\pi_{A_1, A_2, \ldots, A_n} \text{SEM}(N))$$

In our implementation, each `DEFINE TABLE` statement generates a materialized relation, which can then be imported in a DBMS. Alternatively, this relation might also be considered as virtual; this is irrelevant from the user point of view (except for performances).

As an example, consider one of the navigational expressions discussed in the previous section:

```
RoomsPage.RoomList.ToRoom → RoomPage.PaintList.ToPaint →
                    PaintingPage
```

It can be the basis for the definition of a relational view, as follows:

```
DEFINE TABLE AllPaintings (Painter, Title, RoomNumber, RoomName)
AS           RoomsPage.RoomList.ToRoom → RoomPage.PaintList.ToPaint →
                               PaintingPage
IN           UffiziScheme
USING        PaintingPage.Painter, PaintingPage.Title, RoomPage.RoomNo,
             RoomPage.RoomName
```

In this expression, we are essentially giving a name, `AllPaintings`, to a relation corresponding to the navigational expression that includes only a subset of attributes, namely those listed in the `USING` clause, that is, painter name, title, and room number and name.

The relations defined by means of `DEFINE TABLE` statements can be queried by using any relational query language, such as SQL or QBE. For example the query: *"Retrieve all titles of paintings by Botticelli in Room 9"* can be expressed in SQL as follows, by including the view `AllPaintings` in the range list:

```
SELECT    Title
FROM      AllPaintings
WHERE     RoomNo = 'Room 9' AND
          Painter LIKE '%Botticelli%'
```

To give another example, suppose we are also interested in paintings in the *Louvre* virtual museum. Figure 2 shows a fragment of the Louvre Web site: collections are organized in departments, and each department has several sections, each containing some works. Based on the two schemes, we can correlate data in the two sites, by defining views in each of them, and then specifying a query that involves views from different sites (usually joining them). For example, we might ask the following query: *"Retrieve all names of painters having paintings* *in both the Louvre and the Uffizi museums"*. This is done in two steps. First we use two navigational expressions, one for each site, to define views containing all painters names:

```
DEFINE TABLE UffiziPainters (ArtistName)
AS           ArtistsPage.ArtistList
IN           UffiziScheme
USING        ArtistsPage.ArtistList.ArtistName

DEFINE TABLE LouvreArtists (DeptName, Author)
AS           CollectionsPage.DeptList.ToDept →
               DepartmentPage.SectionList.ToSection →
                 SectionPage.WorkList
IN           LouvreScheme
USING        DepartmentPage.DeptName, SectionPage.WorkList.Author
```

Then, we use SQL to correlate the two relations; in particular, we have a join of relation `UffiziPainters` with the selection of `LouvreArtists` that produces the painters, i.e. artists in the paintings department:

```
SELECT     ArtistName
FROM       LouvreArtists, UffiziPainters
WHERE      DeptName= 'Paintings' AND
           ArtistName = Author
```

We would like to emphasize the flexibility and effectiveness of the chosen approach. It is flexible since, once a relational view has been defined and a table has been generated, *any* relational query language can be used to access data. In these examples we SQL, but nothing forbids the use of some more sophisticated language, provided that it can manipulate tables. At the same time, the approach is effective, in the sense that it provides a high-level tool for selecting and correlating data; note that computing the shown queries only by means of browsing and search engines would require a significant effort for the user.

5 Penelope: Building new Structures: Page Restructuring

The approach discussed in the previous subsection is interesting but could be considered as extraneous to the Web framework, where users access information by navigating hypertexts. We thus would like to extend somehow the querying paradigm in such a way that, once data have been retrieved, they are presented to the user as an hypertext. Here, we show how relational views can be transformed back into pages, new pages, with a structure that does not appear in the existing site(s). This technique, called *restructuring*, can be used in two ways: first, as a support to casual queries, where the user wants to browse the results (this could be particularly useful with respect to complex queries with large results);

second, as a means to define a *derived* site, a sort of materialized view over the input sites or over a database.

In order to reach this goal, we introduce a new mechanism, which allows the definition of new page schemes for the query result, according to which data will be organized.

The restructuring process in ARANEUS is composed of three steps. In the first step, the navigations of interest over the base sites are specified and the corresponding relational views are defined, with `DEFINE TABLE` statements. In the second, additional views are defined as needed, in a relational language (say, SQL). In the third, new pages are defined using a specific statement, `DEFINE PAGE`. Let us illustrate the process by means of an example, again on the Uffizi Web site. Assume we are interested in seeing the images of all on-line paintings by Botticelli, each in a page that also contains the title and the date of the painting. More precisely, we could be interested in having *(i)* a page for each painting, with title, date and a large image of the painting and *(ii)* a unique page, with the list of all selected paintings by Botticelli, each with title and link to the respective page. Clearly, these pages have a structure that does not appear in the site scheme, since, for example, titles and (large) images are not included together in a common page. Therefore we need a *restructured scheme* as described in Figure 3; the two page schemes are called `BotticelliPaintingList` and `BotticelliPaintingPage`.

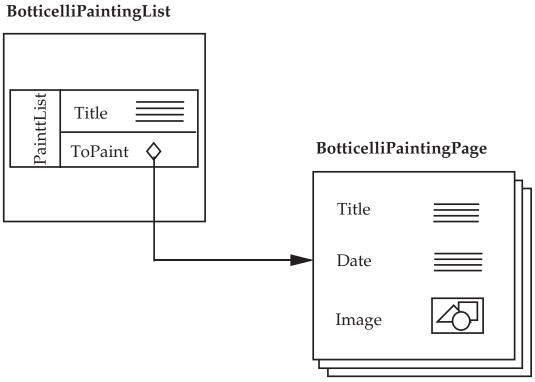

Fig. 3. New page schemes for paintings by Botticelli

In the example, as a first step, we would define a table for the navigations that relate painters with painting images:

```
DEFINE TABLE PaintingsWithImages (Painter, Title, Date, PaintImage)
AS           ArtistsPage.ArtistList.PaintList.ToPaint →
             PaintingPage.ToImage → ImagePage
IN           UffiziScheme
USING        PaintingPage.Painter, PaintingPage.Title, PaintingPage.Date,
          ImagePage.PaintImage
```

Then, we select data of interest, using SQL: we define a view, BotticelliPaintings, over the table PaintingsWithImages, with the standard CREATE VIEW statement.

```
CREATE VIEW BotticelliPaintings AS
     SELECT    Title, Date, PaintImage
     FROM      PaintingsWithImages
     WHERE     Painter LIKE '%Botticelli%'
```

Finally, we specify the structure of the pages to be showed using the following DEFINE PAGE statements. Note that attributes of the source table BotticelliPaintings are enclosed in angle brackets <>.

```
DEFINE PAGE BotticelliPaintingList UNIQUE
AS (URL result.html;
   PaintList: LIST OF (Title:   <Title>;
                       ToPaint: LINK TO BotticelliPaintingPage
                                     (URL(<Title>))));
FROM BotticelliPaintings

DEFINE PAGE BotticelliPaintingPage
AS (URL      URL(<Title>);
   Title:    <Title>;
   Date:     <Date>;
   Image:    <PaintImage>);
FROM BotticelliPaintings
```

These statements generate the HTML code for the new pages, on the basis of the attributes specified. The first DEFINE PAGE statement defines BotticelliPaintingList as a unique page scheme with a multivalued attribute PaintList, corresponding to a list of paintings; note how a local URL, whose value is the string result.html, is assigned to the corresponding instance. Since we declare the page scheme as unique and indicate a single URL, we are assuming that a unique page will be generated by the statement. The second statement defines BotticelliPaintingPage as a page scheme whose attributes are the painting title, the date and the painting image. Here the definition of the URL to be associated with each page is more complex, since it has to be *generated*, and each time we need a *new*, different URL. We use function terms to generate URLs; in fact, term URL(<Title>) specifies that the system has to generate an URL for each page over scheme BotticelliPaintingPage, and that the URL must be uniquely associated with the title value.[9] In this way we can connect the query result page with the associated paintings pages. Here, we are implicitly

[9] This technique is somehow similar to the use of *Skolem functors* to invent new OID's in object-oriented databases [6, 23, 20].

supposing that a different URL can be generated for each painting, i.e. that the title uniquely identifies the painting.

We now want to define more precisely the semantics of `DEFINE PAGE` statements. For the sake of space, the development will be rather informal. The main idea, here, is to create pages starting from tuples in (nested) relations. We first need to introduce two important concepts, namely *local URLs* and *structures*.

Local URLs are local file names used to identify pages; they can be either constant strings, or strings built using the function symbol `URL` from attributes in relations. For example `result.html` is a constant local URL, whereas `URL(<Title>)` denotes a local URL built from values of attribute `Title`; function `URL` generates a different (and new, that is, not already used) file name for each different value of the attribute. The other important concept are *structures*. They are very similar to types, in the sense that describe how page structures can be created starting from attributes in relations.

A `DEFINE PAGE` statement has the form:

```
DEFINE PAGE  P [UNIQUE]
AS           S
FROM         R
```

where: *(i)* R is a relation; *(ii)* P is a page scheme name; and *(iii)* S is a structure built using attributes of R. The `UNIQUE` keyword is optional; intuitively, it is used to specify that the defined page scheme is unique.[10] The semantics of these statements can be informally defined as follows:

- relation R is first extended by adding local URLs to each tuple as new attributes; first, the page URL is generated and added to the relation as a new attribute, `URL`; then, for any structure of the form LINK TO P (*LocalURL*), an attribute `ToP` is added to the relation; for each of these attributes, if the function term `URL()` has been used, a file name is generated for each tuple in the table; we also suppose that attributes are renamed accoding to the new page structure; consider for example the `DEFINE PAGE` statement in the previous example:

```
DEFINE PAGE BotticelliPaintingList UNIQUE
AS (URL result.html;
   PaintList: LIST OF (Title:   <Title>;
                       ToPaint: LINK TO BotticelliPaintingPage
                                     (URL(<Title>))));
FROM BotticelliPaintings
```

Suppose `BotticelliPaintings` is the following relation:

[10] To avoid inconsistencies, we have to impose some constraints for unique page schemes; for example, the corresponding local URL must be a constant string; second, the associated page structure must contain a single attribute, which has to be multivalued.

`Title`	`Date`	`PaintImage`
The Return of Judith	1472 circa	botgiudi9.jpeg
The Discovery of the Murder of ...	1472	botolo9.jpeg
...	...	...
Sant'Ambrogio Altarpiece	1467-70 circa	bottamb10.jpeg

Adding local URLs yields the following table, in which URLs for painting pages are generated by the systems from values of the `Title` attribute:

`URL`	`Title`	`Date`	`PaintImage`	`ToPaint`
result.html	The Return...	1472 circa	botgiudi9.jpeg	returnjudith.html
result.html	The Discovery...	1472	botolo9.jpeg	discoveryhol.html
...	...	...	...	...
result.html	Sant'Ambrogio...	1467-70 circa	bottamb10.jpeg	ambrogioaltar.html

- this relation is then projected onto the attributes occurring in S, plus the URL; since only the title and the link to `BotticelliPaintingPage` are reported in the `DEFINE TABLE` statement, the following table is produced:

`URL`	`Title`	`ToPaint`
result.html	The Return of Judith	returnjudith.html
result.html	The Discovery of the Murder of ...	discoveryhol.html
...	...	...
result.html	Sant'Ambrogio Altarpiece	ambrogioaltar.html

- next, the projected relation is *nested* [30] according to structures in S; more specifically, for each multivalued structure in S, the relation is nested on the corresponding attributes; with respect to the example, a new attribute `PaintList` is introduced by nesting with respect to `Title` and `ToPaint`; the new relation is the following:

`URL`	`PaintList`	
	`Title`	`ToPaint`
result.html	The Return of Judith	returnjudith.html
	The Discovery of the Murder of ...	discoveryhol.html
	...	...
	Sant'Ambrogio Altarpiece	ambrogioaltar.html

- finally, a local page is generated for each tuple in the resulting relation. Nested attributes are conventionally translated using HTML list environments. Note that, coherently with the fact that the page scheme has been defined as being unique, the nested relation for `BotticelliPaintingList` contains a single tuple, and thus a single page is generated.

A similar algorithm can be used to generate the pages for `BotticelliPaintingPage`. In such a case, however, no nesting has to be performed, so the final table has several tuple, one for each painting, and different pages are generated.

6 Discussion

We briefly comment on some of the choices and limitations of our model, and on the main features of the current implementation.

First of all, it is important to stress that in this paper we refer to actual Web sites, over which we have no control: by means of the model one can abstract properties that are of interest from his/her point of view, and different schemes could be built from the same site. The model is not aimed at representing all HTML features, but only those that we found relevant in modeling repeated patterns and interrelated pages.

The model has a lot of similarities with complex-object models with object identifiers (OIDs) [6, 25], with some differences and restrictions. Let us comment on two aspects. First, it is clear that URLs play here a role that is similar to that of OIDs in complex-object models. However, there is one difference: OIDs are completely transparent, whereas URLs are visible and, although in most cases they are used just as references, it is possible to make use of their actual structure and values (for example, they can be examined to check physical location over sites). The second aspect worth mentioning is that complex-object models usually have various constructors: set, list, multiset (or bag), tuple. Here, we have a first level structure that is a tuple, and then we have only a list constructor (whose components are in turn tuples). The choice is motivated by the fact that single elements in an HTML page can be singled out as simple attributes or as elements of a list.[11] HTML does not contain anything similar to set or multiset. HTML tables can be modelled by means of our lists as well.

In the current implementation, in order to see pages as instances of page schemes and extract attribute values, we wrap them using Java classes. Every page scheme in the site corresponds to a specific class with one method for each attribute. Attribute values are extracted from the HTML source using a *text restructuring* language. For instance, the name of a painter, i.e. the value of the `Painter` attribute of page scheme `PaintingPage`, is marked up by tags `<b>` and `</b>`, while the title of the picture, i.e. the value of the `Title` attribute, is marked up by tags `<i>` and `</i>`. Different tools (see, for example, [17]) can be used for text extraction. In the Araneus System we use *Editor Programs* [10], a formalism for text extraction and restructuring. Given a page scheme, each method in the corresponding class is essentially an editor program accessing the HTML source and returning a complex value for the attribute. In essence, the class acts as a *wrapper* for the site pages.

We see each page as a *nested relation* [5, Chapter 20], [8, Chapter 8], in which list attributes are modeled using tables. Due to the absence of duplicates, our relations can be decomposed in flat relations.[12] Based on this perspective, the semantics of navigational expressions can be easily defined using joins. To do this, for each page in a navigation, we generate the associated table, and then

[11] For example, defined by means of the possibly nested `<li>` construct in HTML.

[12] This is due to the fact that we suppose nested structures to be in *Partitioned Normal Form (PNF)* [30]

join them using a local SQL engine. This is also used to query the resulting relational view.

Navigational expressions are somehow similar to *path expressions* [22] in complex-object databases, with an important difference. Since the scheme describes a virtual database, we do not have access methods to page schemes. Thus, to evaluate a navigational expression, we have to start from a unique page whose URL is known, and actually navigate a site.

There are a few important features (that is, HTML constructs) that we have not considered here, but we believe that their relevant aspects could be incorporated in our model with a reasonable effort. Again, we comment on the two aspects we consider most important. First, we have not modelled the fact that HTML links can refer (by means of *named anchors*) to specific locations in pages, rather than to the page as a whole. We could model this by allowing edges in our schemes to have a head that is not just a page (or the highest-level tuple of it), but any component in its structure. However, this aspect is not essential from our point of view. Second,and more important, we have not discussed here a widely used HTML construct, namely *forms*. Forms essentially allow to "select" data specifying some parameters; usually, in response to the submission of a filled-in form, the user receives an HTML page that is not physically stored in the site but is *dynamically* generated. In Araneus, we consider these dynamic pages in the same way as static ones, and forms as "virtual" lists that cannot be scanned but only directly accessed by means of the specification of parameters. The extension involves some details that cannot be included here for the sake of space, but can be carried out with the same philosophy.

Acknowledgments We would like to thank Salvatore Labonia who did a great job in implementing many features of the prototype.

References

1. The araneus Project Home Page. `http://poincare.inf.uniroma3.it:8080/-Araneus`.
2. The Louvre Web server. `http://www.louvre.fr`.
3. The Uffizi Web server. `http://www.uffizi.firenze.it`.
4. S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *International Conf. on Very Large Data Bases (VLDB'93), Dublin*, pages 73–84, 1993.
5. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison Wesley Publ. Co., Reading, Massachussetts, 1994.
6. S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD International Conf. on Management of Data*, pages 159–173, 1989.
7. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. `http://www-db.stanford.edu`, 1996.
8. P. Atzeni and V. De Antonellis. *Relational Database Theory: A Comprehensive Introduction.* Benjamin and Cummings Publ. Co., Menlo Park, California, 1993.

9. P. Atzeni, A. Masci, G. Mecca, P. Merialdo, and E. Tabet. ULIXES: Building relational views over the web. In *Thirteenth IEEE International Conference on Data Engineering (ICDE'97), Birmingham, UK*, 1997. `http://poincare.inf.uniroma3.it:8080/Araneus/publications.html`.

10. P. Atzeni and G. Mecca. Cut and Paste. In *Sixteenth ACM SIGMOD Intern. Symposium on Principles of Database Systems (PODS'97), Tucson, Arizona*, 1997. `http://poincare.inf.uniroma3.it:8080/Araneus/publications.html`.

11. P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive Web sites, 1997. In preparation.

12. P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *International Conf. on Very Large Data Bases (VLDB'97), Athens, Greece, August 26-29*, 1997. To Appear. `http://poincare.inf.uniroma3.it:8080/Araneus/publications.html`.

13. T. Berners-Lee, R. Cailliau, A. Lautonen, H. F. Nielsen, and A. Secret. The World Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

14. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *ACM SIGMOD International Conf. on Management of Data (SIGMOD'96), Montreal, Canada*, pages 505–516, 1996.

15. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources. In *IPSJ Conference, Tokyo*, 1994.

16. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *ACM SIGMOD International Conf. on Management of Data (SIGMOD'94), Minneapolis*, pages 313–323, 1994.

17. C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995. Also available as Technical Report n. CS-94-30, University of Waterloo, Department of Computer Science, `ftp://cs-archive.uwaterloo.ca/cs-archive/CS-94-30`.

18. F. Garzotto, P. Paolini, and D. Schwabe. HDM – a model-based approach to hypertext application design. *ACM Transactions on Information Systems*, 11(1):1–26, January 1993.

19. I. S. Graham. HTML *Sourcebook*. John Wiley, 1995.

20. R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Sixteenth International Conference on Very Large Data Bases, Brisbane (VLDB'90)*, pages 455–468, 1990.

21. T. Isakowitz, E. A. Stohr, and P. Balasubramanian. RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 58(8):34–44, August 1995. `http://www.stern.nyu.edu/~ tisakowi/rmm/`.

22. M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *ACM SIGMOD International Conf. on Management of Data*, pages 393–402, 1992.

23. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

24. D. Konopnicki and O. Shmueli. W3QS: A query system for the world-wide web. In *International Conf. on Very Large Data Bases (VLDB'95), Zurich*, pages 54–65, 1995.

25. G.M. Kuper. *The Logical Data Model: A New Approach to Database Logic.* PhD thesis, Stanford University, 1985.

26. L. Lakshmanan, F. Sadri, and I. N. Subramanian. A declarative language for querying and restructuring the Web. In *6th Intern. Workshop on Research Issues*

in Data Engineering: Interoperability of Nontraditional Database Systems (RIDE-NDS'96), 1996.

27. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *International Conf. on Very Large Data Bases (VLDB'96), Mumbai(Bombay)*, 1996.
28. A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. In *First Int. Conf. on Parallel and Distributed Information Systems (PDIS'96)*, 1996. `ftp://db.toronto.edu/pub/papers/websql.ps`.
29. D. Quass, Rajaraman A., Y. Sagiv, J. D. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Fourth International Conference on Deductive and Object-Oriented Databases (DOOD'95), Singapore*, 1995.
30. M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for ¬1NF relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.
31. H. F. Schwartz et al. A comparison of internet resource discovery approach. *Computing Systems*, 5(4), 1992.