Active Hypertext Views on Databases

Gilles Falquet, Jacques Guyot, Luka Nerima*
University of Geneva

Contact: nerima@cse.ucsc.edu

Abstract

Designing and building a Web site aimed to interact with a database can be a demanding task. A way to accelerate this task is to use a declarative language for specifying hypertext views on databases. In this paper, we discuss hypertext concepts for viewing and acting on a database. While viewing a database through a hypertext view enables a user to navigate among interconnected objects, acting on a database consists mainly in finding objects and associating them (either directly or through the execution of a method). Thus, to act on a database from a hypertext view one must be able to link (or unlink) hypertext nodes and to propagate these operations down to the database. We are currently implementing these concepts in the LAZY system.

1 Introduction

The benefits of building and managing a Web site with database technology are well-known. All the commercial DBMS companies offer products to publish the content of databases on the Web. Some of these products allow also to update the database content (e.g. Oracle Web Server, CGI script, Java/JDBC server). The approach is procedural, i.e. consists in writing database programs that generate HTML pages. Other systems provide the automatic generation of HTML pages from the database schema (e.g. O2Web).

An other approach, the declarative one, rests on explicit declarative representations of both data structure and web structure and on the use of a (query) language to translate the first structure into the second one. Building a web site using a declarative language for describing the structure of the site has several advantages. Primarily, it allows to easily create multiple versions of the site over the same data. It also allows to support easily the evolution of both database and web site structures (see [FLM98] for other considerations). Several systems have been developed using a declarative approach. Their aim is to provide Web views over (heterogeneous) data sources (see [WebDB98] and [EDBT98] where many of these systems are described and [FFK+98]). However, to the best of our knowledge, none of these systems produces a view that is updatable, i.e. a view on which one can perform update operations and where these updates are propagated down to the database.

The work presented in this paper is a step in this direction. We extend the LAZY system [FGN98] with update capabilities and with a trigger mechanism. Since our experience with the former LAZY system was positive and that one of the keys of its success was its simplicity, we added the triggering features in a way that matches with the former system and hence only slightly modifies the node specification language.

This paper focuses on the update issues such as insert/replace/delete propagation into the database, link updates and ambiguity management. Although the recovery and concurrency aspects are very important in such a system, we do not address them in this paper.

2 The Models

Building hypertext views consists essentially in mapping database objects to hypertext objects. We chose simple database and hypertext models to base the view definition language on. These models can then be easily mapped to other database and hypertext models to build concrete tools. For instance, we have implemented concrete tools for viewing relational databases and O2 databases on the Web.

---

* Currently on leave at UCSC

2.1 The Database Model

The database model is an object-based model which is a subset of the O2 database model [LRV88][AHV95]. In this model each object has an identity (oid) and a value. We only consider objects whose value are tuples [$name_1$: $value_1$, $name_2$: $value_2$, ..., $name_n$: $value_n$] where each $name_i$ is an attribute name and each $value_i$ is either: an atomic value (of type integer , boolean , float , string , etc.); or a reference to another object; or a collection of references (set, list, bag). Collection values may be manipulated with the classical operations insert and delete (to add or remove an element from a set or a bag), insertAt, deleteAt, etc. (to manipulate lists).

Methods, which describe the behavior of objects, are associated with classes. A method has a name, a signature and an implementation.

A database schema is a set of class definitions - which are composed of a class name, a (structural) type and a set of associated methods - and an inheritance relationship. The type of a class constrains the values of its objects.

In addition, we suppose that a class with type [$a_1 : T_1$, $a_2 : T_2$, ..., $a_n : T_n$] has "setter" methods set_ $a_1(T_1)$, set_ $a_2(T_2)$, ..., set_a_n (T_n) to update the corresponding attribute values.

A database instance is a set of objects and a set of named collections. Each object is an instance of a class and its value belongs to its class type. A named collection is a set (or list) of objects which are instances of a class or of one of its subclasses.

2.2 The hypertext model

We consider a simple hypertext model whose structural part is composed of nodes, anchors, links, triggers and activation rules.

Each *node* has a unique identity and a content. The *content* of a node is a sequence of elements which may be character strings, images, sounds, etc. A *link* is defined by its starting and ending anchors and by its category which is either `reference' or `inclusion'. Reference links are intended to create a navigation structure within the nodes. Inclusion links are intended to create nested structures that represent complex contents (structured documents).

An *anchor* is an element or a sequence of elements within the content of a node, it serves as a starting or ending point of a link.

An *activation rule* is a triple (*event*, *condition*, *action sequence*). When an event occurs, e.g. the opening or the closing of a node, the condition is evaluated and, if satisfied, the sequence of actions triggered. Action is typically a (database or hypertext) method call.

A *trigger* is a set of activation rules associated with a link. In this case, the event occurs each time one navigates through the link .

3 The Hypertext View Definition Language

A hyperview specification consists of a set of node schemes which specify the collection from which the node's content is to be drawn; the selection and ordering criteria; the elements that form the content; and links to other nodes. Since this language is defined in [FGN98] we only give here an informal presentation based on examples. The language constructs related to database update will be presented in the next section.

Node Contents

The content of a node is based on a collection specified in the *from* <*collection*> clause. A sequence of fields specifies how to construct the elements that represent every selected object. For example, let employeesOfDept be defined as:

```
node employeesOfDept[d: Department] is
    "name", emp_name, "title", emp_title,
    from EMP
    selected by dept = d
    ordered by name
```

The content of a node instance employeesOfDept[x] is obtained by

- selecting all the objects in the collection EMP which have the value x for their attribute dept
- ordering these objects according to the values of attribute name
- for each object creating the sequence of elements corresponding to: the string "name", the value of attribute emp_name, the string "title", and the value of attribute emp_title.

Reference Links

The href statement specifies a link. The link's anchor is an element of the starting node which can trigger the navigation to the referenced node. A link specification refers to a node through its schema name and a list of parameter values.

For instance, consider the following node definition:

```
node deptIn[loc: String] is
    no, ": ", name, " ",
    href employeesOfDept [self] " Employees: "
    from DEPT selected by location = loc
    ordered by no
```

(Note: the pseudo variable self iterates over the sequence of selected objects).

The representation of each selected department d will have an anchor text "Employees:" that is the starting point of a link to the node employeesOfDept[d].

Inclusion Links

An inclusion link between two nodes determines a compound-component relationship between these nodes. The target node is to be considered as a sub-node of the source node of the link. This fact should normally be taken into account by the hypertext interface system which should present inclusion links in a particular way (generally by including the sub-node contents within the node presentation). Inclusion links are particularly useful to create multi-level hierarchical nodes to represent complex entities.

4 Updating the database through hypertext nodes

Hypertext nodes are database views. They are intended to simplify the user's or programmer's task by providing them a higher level view of a part of the database. Many effort have been devoted to the definition of views on relational or object-oriented data models [Ber92][SAD94][SLT92].

Hypertext node instances contain elements, mainly atomic attribute values and links. The user will act on these nodes (adding/removing an element of a node; adding/removing a link; creating/deleting a node) and these operations must be propagated in the database by means of traditional database update operations. Since hypertext nodes are views, it is well-known that we will observe ambiguities which are similar to those found when updating relational or object-oriented views [SLT92]. As we will see further, a remarkable benefit in using the hypertext paradigm is that it allows to resolve some of the ambiguities, mainly thanks to the information conveyed by the navigation.

In order to represent the navigation history we will use navigation paths similar to those defined in [MMM97]. But instead of distinguishing three link types (interior, local, and global) we associate with each link its starting anchor text (actually it can be an image's URL, etc.).

A navigation history is a *tree* whose nodes are hypertext view node instances (defined by a name and parameter values) and whose edges are ordered pairs of nodes labeled with anchor texts. The leaves of the tree correspond to the currently open hypertext nodes (note that the same node may be opened more than once).

We denote the history leading to a given node by a path expression of the form

$$node_1[param_1] \text{—}text_1\rightarrow node_2[param_2] \text{—}text_2\rightarrow \ldots \text{—}text_{k-1}\rightarrow node_k[param_k]$$

A navigation history regular expression is a regular expression which denotes a set of navigation paths. Variables may appear in such an expression where parameter values are expected. For instance, the expression

```
deptIn["New York"] -"Emp*"-> employeesOfDept[d]
```

corresponds to all the navigation paths starting at *deptIn["New York"]* and ending at any instance of *employeesOfDept* (*d* is a variable) after following one link whose starting anchor begins with "Emp".

4.1 Inserting simple objects

The insert operation of simple objects (i.e. tuples of atomic attributes) proceeds in two steps: (1) collect the values of the atomic attributes and (2) create a new object in the database.

From the hypertext design point of view, one can use the "delayed include" link to specify that the node devoted to the insert operation must be included as a sub component only after the activation of the anchor text ("New" in the node specification "newEmployeeOfDept" in the example below). The activation of the insert operation is specified by a reference link and a trigger method specification (see last line of the "insertEmp" node specification in the example below).

```
node newEmployeeOfDept[d: Department] is
    include employeeOfDept[d]
    delayed include insertEmp[d] "New"

node employeeOfDept[d: Department] is
    emp_name, " (", emp_title, ")"
    from EMP
    selected by dept = d
    ordered by emp_name

node insertEmp[d: Department] is
    bold("New employee name: "),  bold("Title:"), break(),
    input emp_name: String, input emp_title: String,
    href insertEmp[d] "Insert" triggers EMP.Insert(new Emp(emp_name, emp_title, d))
```

The content of the node instance newEmployeeOfDept[4] (left node in figure 1) is obtained as explained previously. The activation of the link labeled "New" replaces the label with the node instance "insertEmp[4]". The right node of the figure 1 shows the obtained node instance. The activation of the "Insert" link adds a new employee in the underlying collection and goes back to the same node, allowing further insertions.

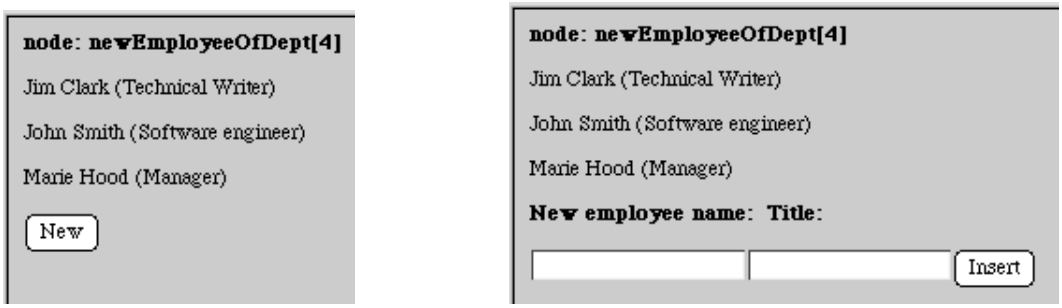

Figure 1: A node instance with a delayed inclusion

4.2 Replacing atomic elements

The substitution of the elements which are atomic attribute values is straightforward: it is sufficient to update the corresponding attributes in the database. For instance, a node meant to enable the element values' replacement can be specified as follow:

```
node replaceEmp[d: Department] is
    "Name:", input emp_name: String, "Title:", input emp_title:String,
    href replaceEmp[d] "update"
```

```
            triggers self.set_emp_name(emp_name);
                   self.set_emp_title(emp_title)
```

4.3 Updating links

The update of a link element depends on the underlying nature of the connection that it describes. For this purpose, we take the structural model similar to [BSK+91], that is (1) m:n connection, (2) ownership connection, (3) reference connection. Actually, only insertion and deletion of links occur in the system. The replacement of a link is a combination of the two operations. Above we present only the first kind of link.

m:n connection. The update of a link that represents such a connection will never produce a deletion of an object in the database. The propagation in the database will be the replacement of the old reference value by the new one. In the example below, we show a specification for a hypertext view that enables to connect mandatory and optional courses to a curriculum:

```
node curriculum[c : Curriculum] is
    heading1(title)
    ...
    "Mandatory Courses:"
    include updateCourses[mandatory]
    "Optional Courses:"
    include updateCourses[optional]
    from CURRICULUMS
    selected by self = c

node updateCourses[s: set Course] is
    include courses[s]
    new href addCourse[s] "add new"

node courses[s : set Course] is
    ...
    href courses[s.delete(self)] "delete" triggers
    if (HISTORY ends with
           curriculum[d] -"Mandatory Courses:"-> updateCourses[sc] -->)
           d.set_mandatory(d.mandatory.delete(self))
    if (HISTORY ends with
           curriculum[d] -"Optional Courses:"-> updateCourses[sc] -->)
           d.set_optional(d.optional.delete(self))

node addCourse[s: set Course] is
    ...
    href courses[s.delete(self)] "add to list" triggers
    if (HISTORY ends with
           curriculum[d] -"Mandatory Courses:"-> updateCourses[sc] -->)
           d.set_mandatory(d.mandatory.insert(self))
    if (HISTORY ends with
           curriculum[d] -"Optional Courses:"-> updateCourses[sc] -->)
           d.set_optional(d.optional.insert(self))
```

4.4 Deleting nodes

The deletion of a node can yield different update operations in the database, depending on the semantics associated with the hypertext view. Thus ambiguities arise.

The aim of the example below is to show that the navigation can help to resolve some ambiguities. In this example, there are three paths to reach an object of the class person:
(1) department -> members -> person
(2) department -> group -> members -> person
(3) department -> soccerTeam -> person

Let us suppose that the user wants to delete the ending node of each of these three navigation paths to express:

- in case 1, that the designated person is living the department. The adequate update operations to perform in the database are to remove the person from the PEOPLE collection and moved it into the FORMERPEOPLE collection;
- in case 2, that the designated person is living the research group. The adequate update operations to perform are to move the person into the FORMERPEOPLE collection, and, either move it to an other group or remove it from the PEOPLE collection;
- in case 3, that the person is no longer in the soccer team. The appropriate operation to perform in the database is to set the person's attribute *soccer_team_pos* to "".

The nodes can be specified as follows:

```
node department[d: Department] is
    "Welcome to the ",dept," department",
    "About the department:", about,
    href members[dept_members] "Members",
    "Research groups:",
    include groupNames[self]
    ...
    href soccerTeam[d]
    from DEPARTMENTS
    selected by self = d

node groupNames[d: Department] is
    href group[self] name
    from GROUPS
    selected by dept = d

node Group[g: Group]
    href projects[group_projects] "Projects"
    href members[group_members] "Members"
    href formerMembers[self]
    from GROUPS
    selected by self = g

node members[p: set Person]
    href person[self] name, phone, office, email
    from PEOPLE
    selected by self in p

node formerMembers[g: Group] is
    href person[self] "name", name, "current address", address
    from FORMERPEOPLE
    selected by group=g

node soccerTeam[d: Department] is
    href person[self] "name", name, "position:", soccer_team_pos
    from PEOPLE
    selected by dept=d and soccer_team_pos ≠ ""

node person[p: Person] is
    name, ...
    new href person[self] "delete" triggers
    if (HISTORY ends with
           department[d] - "Members" -> members[s] - name ->
          person[p] -->) p.moveToFormerPeople()
    if (HISTORY ends with
           department[d] - "Research groups:"-> groupNames[d] - grname       ->
    group[g] - pname -> person[p] -->) ...
    if (HISTORY ends with
           department -> soccerTeam -> person -->)
```

```
      self.set_soccer_team_pos("");
from PEOPLE
selected by self = p
```

5 Conclusion

We have shown that updating databases through hypertext views can take advantage of the navigational nature of such views, in particular to disambiguate operations which create or delete relationships between objects. In order to exploit hypertext navigation for this purpose we model the navigation space as a tree which can be queried by regular expression matching.

Implementation is underway, based on a node server (a Java Servelet which accesses relational databases through JDBC, maintains the navigation history and manages the execution of triggers). We are also exploring the generation of CORBA components from node specifications.

Future Work. We are studying the design of active hypertext views. In order to devise design patterns, we will use the developed translator to implement active views on an existing database. We will then compare these views with the existing interfaces in terms of expressive power and specification complexity. Automatic generation of updatable views based on the database schema will also be part of our study. These basic views can then be improved by applying a sequence of refining operators. Our overall goal is the creation of a library of design patterns.

Another important task that we plan is to include transaction monitoring to our system.

Practical applications. Obviously, there is a huge potential for practical applications of active hypertext view generation tools. Among them one can cite:

Active navigable catalogs:

Database query languages are not well suited for handling vague or imprecise queries or for browsing through data. A navigational interface, on the contrary, can help a user whose needs cannot be precisely stated. Once the user has found what he or she was looking for, the active hypertext enables him to take any appropriate action. For instance, he should be able to order an item found in an electronic catalog. He could also virtually build a customized product by assembling different parts, etc. The active hypertext view specification language could thus be used as a mean to specify the interaction between a customer and a "virtual store".

Specification of public access to sophisticated services:

Any interactive service offered to non specialist users, or to the public, which is not limited to browsing and querying, needs a way to act on persistent data (usually stored in a database). For instance, a university could offer on-line services such as registration, enrollment in courses, etc. Web shopping cart like applications constitute also ideal applications.

Rapid development of sophisticated IS interfaces:

It is well known that the development of the user interface of an information system is a time consuming task. Moreover, the design of effective and efficient interface is a complex task which requires skills that are hardly found among developers. Active hypertexts, augmented with simple manipulation operations, offer an interface paradigm which is usually well understood by users. Thus they can be used to rapidly develop and test user interfaces.

References

[AHV95] 1.S. Abiteboul, R. Hull, V. Vianu. Foundations of Databases , Addison-Wesley, 1995.

[BSK+91] Barsalou, N. Simabela, A. Keller, G. Wiederhold. "Updating Relational Databases through Object-Based Views". In Proc. ACM SIGMOD, Denver, pages 248-257, 1991.

[Ber92] E. Bertino. "A View Mechanism for Object-Oriented Databases". In Proc. EDBT 92 conf., pages 136-151, 1992.

[BMP+92] P. Borras, J. C. Manou, D. Plateau, B. Poyet, D. Tallot. "Building User Interfaces for Database Applications: the O2 Experience". SIGMOD Record, Vol. 21, No. 1, pages 32-38, 1992.

[EDBT98] Proc. of the EDBT'98 Conf., Valencia, 1998.

[FGN98] G. Falquet, L. Nerima, J. Guyot. "Language and Tools to Specify Hypertext Views on Databases". In Proc. of the WebDB Workshop, Valencia, 1998.

[FGN96] G. Falquet, L. Nerima, J. Guyot. "A Hypertext View Specification Language and its Properties". CUI Technical report #102, University of Geneva, 1996.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Levy, D. Suciu. "Catching the Boat with Strudel: Experiences with a Web-Site Management System". In Proc. ACM SIGMOD Conf., Seattle, pages 414-425, 1998.

[FLM98] D. Florescu,, A. Levy, A. Mendelzon. "Database Techniques for the World-Wide Web: A Survey". ACM SIGMOD Record, Vol. 27, No. 3, Sept. 1998.

[FS92] H. P. Frei, D. Stieger. "Making use of Hypertext Links when Retrieving Information". In ECHT 92, ACM, pages 102-111, 1992.

[LRV88] C. Lécluse, P. Richard, F. Velez. "O_2, an Object-Oriented Data Model ". In Proc. ACM SIGMOD, Chicago, 1988.

[MMM97] A. Mendelzon, G. Mihaila, T. Milo, Querying the World Wide Web, Int'l Journal on Digital Libraries Vol. 1, No. 1, pages 54-67, 1997.

[SAD94] C. Santos, S. Abiteboul, C. Delobel. "Virtual Schemas and Bases". In proc. extending Database Technology Conference (EDBT), 1994.

[SLT91] M.H. Scholl, C. Laasch, M. Tresch. "Updatable Views in Object-oriented Databases". In Deductive and Object-oriented Databases --- DOOD'91, pages 189-207, 1991.

[WebDB98] Proceedings of the WebDB 98 Workshop, Valencia, 1998.