Hierarchical Views of Databases

Joe Edelman
Dartmouth College *
joe@orbis-tertius.net

March 1, 2004

Abstract

Relational, semantic, and hypertext databases are increasingly common. Often these databases contain a small number of primary data types (e.g., patrons, books) and a large number of ancillary data types (e.g., addresses, account histories, publishers, etc.), but generic database user-interfaces do not take advantage of this fact. We present techniques for identifying this small set of primary data types within a database and for using it to generate intuitive, generic, hierarchical user-interfaces for complex databases. These interfaces have many of the usability and semantic advantages traditionally associated with file systems, yet can provide access to richly interconnected data models. A modular open-source implementation based on a meta-object protocol is discussed.

1 Motivation

This paper is concerned with generic database interfaces, rather than custom-built, domain-specific interfaces. Prior work in the field can be divided in three broad categories: there are linguistic, "command-line" interfaces (e.g. SQL, Prolog, Smalltalk, CLOS), there are graphical, forms-based interfaces (e.g. FileMaker Pro, MS Access, Protege, Mogetto), and there are graph-based and spacial database visualization techniques.[1–4]

In contrast, we present an interface that is tree-based. This interface, described in sections 2 and 3, seems to have several advantages over prior systems: Unlike graph-based and spacial visualizations, this interface can be displayed and manipulated with the standard, familiar widgets used for exploring and reorganizing filesystems and other hierarchical structures. Unlike linguistic interfaces and forms-based interfaces, this interface can provide a top-down view of the entire database, simplifying navigation and reducing the need for prior understanding of the data model.

*This work was done at the fMRI Data Center under the direction of Jack van Horn and Jeff Woodward, and supported by NIH grant X and NSF grant Y.

The most important advantage, however, is that the tree-based interface allows the user to reuse a *conceptual model*—concerning folders, objects, attributes, and attachments—that is more familiar than the query/command models of linguistic systems, the linkage model of forms and hypertext tables, or the map/chart models of visualization techniques. This is covered in section 4.

In section 6 we describe the architecture of the application itself, which is written in Java, freely available, and structured so as to be easily applied not only to relational databases, but also to hypertext stores, object-, XML schema-, and description logic-based databases.

2 Primary Data Types

During execution of a task, users will often discuss a database as if it contained only one type of data. For example, CiteSeer[5] is referred to as a "citation database" while following references and an "article database" when looking for a particular paper. It is rarely if ever referred to as an "author database" or a "sentence database", although it is also these things. Generally these data types play supporting roles.

Our approach to database interaction is based on these observations: that during a particular task, one data type is often *primary* in users' minds, and that some data types are much more likely to be considered primary than others. The choice of primary data type is a major component of the *conceptual model* (see section 4) with which the user approaches the database, and can be used to construct a generic interface which is nonetheless specific in some ways to the user's task.

Given a particular database, those data types which are most likely to be considered primary we will call *primary types* (e.g. articles and citations in CiteSeer). Data types which are unlikely to be considered primary we call *ancillary types* (e.g. authors, sentences, dates, books). For any particular task, we will consider the *selected primary type* (or just *selected type*) to be the particular primary type in the user's mind as he performs that task.

Generally, even in databases with thousands of types, the number of distinct primary types is small. Our database navigation framework, *Knowledge Explorer*, uses an algorithm (described in section 5) to identify this small set of primary types in a database, and allows the user to select among them as their task changes. During any particular task, the selected primary type is used to generate a hierarchical view of the database, where objects of non-selected types are visualized as attachments to the objects of the selected type.

3 A Tree-Based Interface

Figures 1 and 2 show screenshots of this interface. On the left side of the window is the *tree pane*, and on the right is the *content area*. The tree pane always displays a reconfigurable, folding tree view of the database. The content area

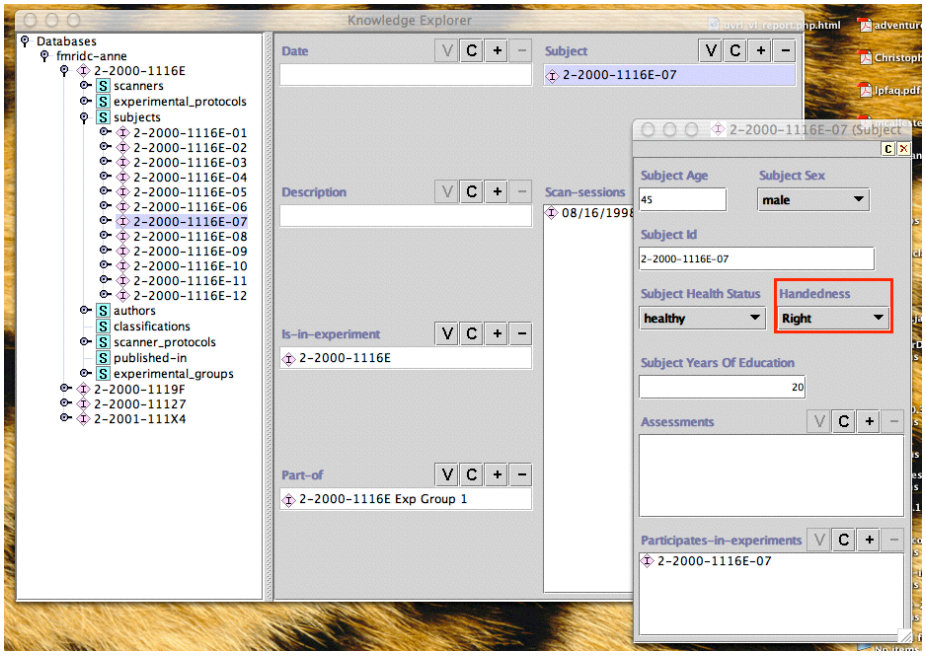

Figure 1: The Knowledge Explorer. The content area is displaying a form. This database concerns MRI radiology experiments. The selected type is 'experiment' (identified by a serial number), and we are viewing the form for a subject in a particular experiment.

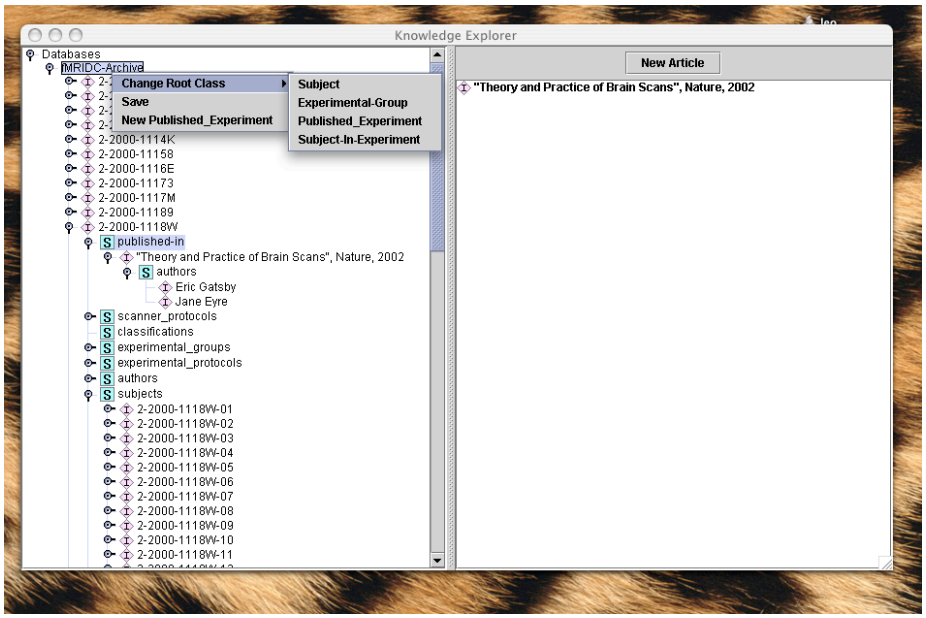

Figure 2: The Knowledge Explorer. The content area is displaying a list, so as to facilitate drag-and-drop to another part of the tree. The contextual menu for selecting an alternate primary type is shown.

can either display the contents of a virtual "folder" from the tree view (figure 2), or a hypertext-form view of some information object (figure 1). Objects can dragged and dropped between panes. The entire interface is designed to be similar to the Windows File Explorer, a familiar and convenient interface for browsing, manipulating, and operating upon data in trees.

3.1 Tree Pane

It is in the tree pane that the user does broad, hierarchical exploration of the database. Each object of the selected type appears in the tree pane under a common root that represents the database itself. These objects have standard disclosure triangles which reveal the one-to-many relations they participate in with objects of non-selected types. These one-to-many relations, which we call *attachments*, are distinguished in the user interface from one-to-one relations and many-to-one relations—the later we call *attributes*, and they are not listed in the tree pane, but only in the content area. Only databases, objects of the selected type, their attachments, their attachments' attachments, etc., are listed in the tree pane, and any particular object's attachments are grouped by the relation through which they are attached. The user may either "drill down" into these relations in the tree pane or click on them to reveal the set of related objects and possible operations on them in the content pane.

By default, the "most primary" data type (see section 5) is the selected type when *Knowledge Explorer* is started. By right-clicking on the root item in the tree, the user can select an alternate primary data type at the root of the tree. This is done with a contextual menu (figure 2), so as not to distract new users who wish only to use the default view, and the user need only select from a short list of primary types in the particular database they are viewing.

Other notable features of the tree pane include the use of typedown as a limited query system, and the Windows Explorer-like auto-expansion of tree nodes based on actions in the content pane.

3.2 Content Area

The content area can either display the contents of a virtual "folder" from the tree view (figure 2), or a hypertext-form view of some information object (figure 1). In the former case, it acts as an extension to the tree pane, allowing the user to view two arbitrary objects' relations side-by-side, and to move or copy attachments between them. The content area can also display a button bar offering a set of actions, such as creation of attachment objects of various types, in this mode.

In the latter case—when the user has selected a particular data object in the tree pane—the content pane acts like a hypertextual form in the tradition of Mogetto[?] and *Protégé*.[?] In fact, we actually call out to the open-source *Protégé* toolkit (see section 6) to generate these forms. The forms that *Protégé* generates are quite sophisticated, and support their own kind of hypertextual navigation, which we have integrated with the tree pane: where instances appear in the

content pane, they can be double-clicked to "jump" the tree view to wherever that instance is rooted or, if option is held, that instance's form can open in a new window, and the tree view is undisturbed.

4 Advantages of Task Sensitive and Tree-Based Interfaces

Despite the greater generality and power of modern database and hypertext systems—and the oft-made[6–9] observation that these systems model the real world and the nature of thought better than hierarchical systems—users seem to prefer to manage files and explore structures that are arranged in trees. It is enough to compare the popularity of graph-structured Choose-Your-Own-Adventure books, wikis,[?] and programming environments like Squeak,[?] on one side, with books that sport a standard table of contents, file systems, and rigidly hierarchical programming environments,[?] on the other. Tree-like structures are used except when, in cases like the world-wide web or large transactional databases, a tree organization is clearly impractical.

We can begin to account for this by investigating the information available in tree-based structures for *exploring*, *finding*, and *manipulating*, and why this information cannot easily be made available in more general structures. We find that tree structures offer a *top-level view* and cues about the *promise* of whole sections of the tree which are of use in exploring and do not generalize to arbitrary relational structures. Similarly, the notions of *proximity* and *direction* which operate in trees offer cues for *approach* when a user is trying to find a particular piece of information. Lastly, a notion of *containment* which is fundamental to tree structures conveys information which greatly simplifies all three activities which we are considering: exploration, search, and manipulation.

Exploration, with data structures as with caves, implies a *limited view* together with a set of *options* for extending or translating that view. The explorer must identify which options will extend their view in the most interesting or informative ways. Within tree structures, the process of exploring consists of *scrolling* and *unfolding* a graphical widget or, in conversational systems like the unix shell, of issuing directives that alter and describe the working directory. In either case, the user is presented with a level or two of the tree and chooses either to move upward in the tree, towards more general levels of organization, or downward, unfolding or describing a particular node. At each level of hierarchical organization, there is a clearly-defined collection of options, all of the same type, and the data structure from that point down is *summarized* by the options available. Since each tree node is the root of an entire section, and since the views available under tree nodes are disjoint, the merits of exploring each section can be considered separately, and if none of them are found appealing, the user may retreat to a higher level, where there will be yet another top-level view.

Exploring a graph based structure is difficult one never knows whether the

links one sees are disjoint, or how well the links available behind them are summarized by their label

interfaces like touchgraph help remedy this situation in two ways: (1) touchgraph will 'scout out' a link for you, visualizing links available beyond each link available for the current object; (2) touchgraph does a crude kind of multidimensional scaling on the graph in question, and as a result creates visual *clusters* of tightly linked or related items; the user can try to evaluate the promise of clusters depicted even when it remains impossible to evalulate the promise of sections.

...

Finally, it is extremely satisfying to use a generic interface which adapts itself to the task at hand.

5 Algorithms and Definitions

In this section we formalize the distinction between primary and ancillary data types made in section 2, then present the scoring function which we use to identify primary types in our software. We will begin, however, with a characterization of databases and data types in general. This characterization serves as the model underlying our software's back-end (see section 6) and is broad enough to include many types of databases, including relational, object-oriented, hypertextual, semi-structured, and logic-based.

Definition 1 *A database $D = (O, T, R)$ is defined as a set of objects, a set of types, and a set of relations. There is a correspondence between objects and types, such that for each type $t_i \in T$ we have a set of objects $\{o_1, o_2, \ldots\} \in O$ of that type. Each relation $r \in R$ is an association among objects of various specified types*

For example, the CiteSeer database may contain types corresponding to authors, papers, citations, numbers, and strings, and a relation "name" might connect objects of type author with objects of type string.

We will first consider the set of types T in the database. We say that the types of a database are either primary types or ancillary types, i.e. $T = T_P \cup T_A$. In section 2, we described ancillary types as playing *supporting roles* in a database. By this we mean that objects of ancillary type are added to a database so as to more fully describe objects of primary type. For instance, objects of type *publisher* may be added to more fully describe the *books* in a library's database. Unlike many other type distinctions that have been made, [1] there is no *a priori* way to distinguish primary from ancillary types. For example, it may be that in CiteSeer the ancillary type *author* was included so as to describe objects of primary type *paper*, while in some more biography-oriented database, the ancillary type *paper* could be subordinate to a primary

[1] see, for example, the distinction between printable and abstract types[10] or between informational and referential types.[?]

type *author*. The concepts of paper and author are the same; what differs—and determines their categorization—is their role in the database.

For this reason, it is unlikely that by paying attention only to the database *schema*—which concerns the conceptual relationships in which objects of the various types can engage—we could classify types as either primary or ancillary. Instead, we take summary statistics on how actual instances of each type tend to be connected to other instances in the database: First, objects of ancillary type are more likely to be *simply connected* to other objects, i.e., they are more likely to participate in one-to-one or one-to-many relationships and less likely to participate in many-to-many maps. Secondly, it is unlikely that there are objects of ancillary type that are not reachable via some object of primary type, i.e., objects of primary type are likely to *cover* the database. Finally, objects of ancillary type are likely to serve either as attachments or labels on objects of primary type—in which case there will be dramatically more of them—or as classifier for objects of primary type—in which case there will be much fewer; that is, objects of ancillary type occupy either tail of the population distribution of types.

In our *Knowledge Explorer* implementation, we capture these notions with a few simple scoring functions which operate on the population sizes of types, the number of objects reachable through their relations, and the complexity with which a median object of a given type is connected to other objects. These functions use arbitrary constants that seem to work well, but clearly a superior system could be constructed based on a deeper statistical analysis or a deeper understanding of these issues than we presently have.

Definition 2 *We define a* good-sized type *as one whose population neither has too few objects in it nor too many. This has the double advantage that we do not choose classifier types or attachment types as primary, and that we give the user a moderately-sized set of navigational choices, without overloading the user-interface.*

In our implementation we define a logarithmically-interpolated scoring function (table 1) for whether the population of a type t_i is good-sized.

Table 1: Scoring function on population size. A score of 1 indicates that type is good-sized.

$\lvert t_i \rvert$	$P(t_i)$
0	0
5	1
1,000	1
10,000	0

Functions concerning reachability and complexity of relationship are based on median cardinality:

Definition 3 *A relation r has* median cardinality $c(r, t_i) = n$ *with respect to a type t_i, if, over all data objects $o \in t_i$, the median number of other data objects connected to o via r is n. Mathematically,*

$$c(r, t_i) = \text{median}_{o \in t_i} |r(o)|$$

For example, the median cardinality of the relation *lives-in-house* with respect to the type *human* is 1.

Definition 4 *An* exciting *relation r with respect to a type t_i is one whose median cardinality is neither too large nor too small. Intuitively, we want exploration of r through the unfolding of tree nodes under each $o \in t_i$ to be rewarding, but not overwhelming.*

In our implementation we define a scoring function for whether a given relation r is exciting with respect to a type t_i,

Table 2: Scoring function for median cardinality of a relation

$c(r, t_i)$	$E(r, t_i)$
1	0
12	1
1,000	0

Finally,

Definition 5 *An* primary *type t_i is a type with a good-sized population and at least two complex relations, i.e.,*

$$T_P = \left\{ t \in T | P(t_i) \cdot \sum_{r \in R} E(r, t_i) \geq 2 \right\}$$

All such primary types become candidates for selection in the tree pane's contextual menu, and the type with the highest $P(t_i) \cdot \sum_{r \in R} E(r, t_i)$ value is the default selected type when a new database is opened.

6 Meta Object Protocol

Definition 6 *A* typing *of a database G is a set of types $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ such that each data object $o \in V$ in the database is in at least one of the types of $\mathcal{T}$. The data objects within a type t_i are called its* population.

In our implementation, we are interested specifically in typings that are meaningful to end-users, i.e., where the members of each type $t_i \in \mathcal{T}$ have something in common with each other.

We can obtain such a typing in a variety of ways: for object-oriented databases and most knowledgebases, there is information within the database itself about the types of data objects, and this can be used. If this information is missing, as may be the case in hypertext or in relational databases, or if it is incomplete, a typing can be provided by plug-in code (see section 6.2) or by an OWL type classifier[?] that uses qualitative information in relations or attributes to assign each object to a set of types.

6.1 java code - lost section

*Knowledge Explorer*is a Java application implemented according to object-oriented principles. Use of the Java Swing framework and the *Protégé*family of data and knowledge representation tools allows it to be quite small—less than 1,500 lines of code—and anyone familiar with Java could readily understand its structure and make modifications. It is hosted on SourceForge,[?] and has already become quite popular.

6.2 *Protégé*: Database Interface

Integration of *Knowledge Explorer*with various data-, object-, and knowledge-bases is via *Protégébackend plugins*. Plugins exist for reading and manipulating data in a variety of formats, including RDF, OWL, XML + XML Schema, CLIPS, and the formats used by various Prolog-like reasoning engines. Creating of a backend plugin for an arbitrary relational or object schema involves implementation of a simple Java API (Figure 7) which serves three purposes: First, for data models which are not self-describing (i.e., for relational models where the set of entities and relationships is not actually stored within the database) this information needs to be encoded in Java. Second, the semantics of the underlying data model and the allowed operations upon the data must be specified. Third, the actual methods by which entities (instances) can be retrieved and their attributes (slots) looked up and redefined must be declared. For most object and relational databases, this is a straightforward process, and there is some documentation about how to do it on the web.

One advantage of the use of *Protégé*is that, through the mechanism of *project inclusion*, it is also simple and straightforward to agglomerate multiple databases

6.3 *Protégé*: Forms

The forms interface in *Knowledge Explorer*is also implemented through the *Protégé*Forms API. Not only are the forms automatically generated—they are based on the same database schema and type information that drives *Knowledge Explorer*'s hierarchical view. If necessary the user can customize or override these forms as well. Simple rearrangements and widget substitutions can be done with the *Protégé*software, or entirely new widgets and forms interfaces can

be implemented in Java, installed in the Java classpath, and will be recognized and loaded in *Knowledge Explorer* as necessary.

7 Conclusions

Many questions are left unanswered by this work. For instance, although user response to the application has been extremely positive, it is unclear exactly which kinds of databases and tasks are most appropriate for a hierarchical view. It is also likely that there better methods for the production of such a view, and that there are applications of the conceptual model beyond the folding tree graphical user interface (i.e., in queries or speech, or tangible media).

It has been 25 years since the adoption of relational databases[7] and 13 years since the popularization of hypertext,[8] but most people remain unwilling to keep their local documents in a database or a hypertext store (e.g. wiki). The dominant organizational scheme, even among programmers, remains one of files and folders. As the amount of digital information in our lives grows dramatically, this situation becomes more and more limiting and painful. It is hoped that this paper can contribute to the spread of more sophisticated information management solutions.

References

[1] Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, /2000.

[2] Kenneth J. Goldman, Sally A. Goldman, Paris C. Kanellakis, and Stanley B. Zdonik. ISIS: interface for a semantic information system. In Sham Navathe, editor, *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data, May 28–31, 1985, LaMansion Hotel, Austin, Texas*, pages 328–342, New York, NY 10036, USA, 1985. ACM Press.

[3] David Huynh, David R. Karger, Dennis Quan, and Vineet Sinha. Haystack: a platform for creating, organizing and visualizing semistructured information. In W. Lewis Johnson, Elisabeth André, and John Domingue, editors, *Proceedings of the 2003 International Conference on Intelligent User Interfaces (IUI-03)*, pages 323–323, New York, 12–15 2003. ACM Press.

[4] Peter Sawyer and John A. Mariani. Database systems: Challenges and opportunities for graphical HCI. *Interacting with Computers*, 7(3):273–303, 1995.

[5] Steve Lawrence. Online or invisible? *Nature*, 411(6837):521, 2001.

[6] Donald A. Norman. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine.* Addison-Wesley Publishing Perseus, Reading, Massachusetts, 1993. ISBN 0-201-62695-0.

[7] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[8] Tim Berners-Lee. *Information Management: A Proposal*. CERN DD/OC, Geneva, CH, 1989.

[9] Ole-Johan Dahl and Kristen Nygaard. SIMULA, an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[10] Richard Hull and Roger King. Semantic database modeling: survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.