Application of Information Technology ■

WebEAV:

Automatic Metadata-driven Generation of Web Interfaces to Entity-Attribute-Value Databases

Prakash M. Nadkarni, MD, Cynthia M. Brandt, MD, Luis Marenco, MD

Abstract The task of creating and maintaining a front end to a large institutional entity-attribute-value (EAV) database can be cumbersome when using traditional client–server technology. Switching to Web technology as a delivery vehicle solves some of these problems but introduces others. In particular, Web development environments tend to be primitive, and many features that client–server developers take for granted are missing. WebEAV is a generic framework for Web development that is intended to streamline the process of Web application development for databases having a significant EAV component. It also addresses some challenging user interface issues that arise when any complex system is created. The authors describe the architecture of WebEAV and provide an overview of its features with suitable examples.

■ **J Am Med Inform Assoc.** 2000;7:343–356.

The entity-attribute-value (EAV) physical database architecture is widely used in clinical data repositories (CDRs). Those CDRs with a major EAV component include the pioneering HELP system[1,2] (and its commercial version, the 3M CDR[3]) and the Columbia–Presbyterian Medical Center CDR.[4,5] Entity-attribute-value design addresses a problem that conventional table design (i.e., one column per finding or parameter) cannot address. Specifically, data on several thousand potential parameters can be stored for a patient across all clinical specialties. If these data are modeled as one database field per parameter, numerous tables are required to hold the data, and these tables will require repeated modification as medicine advances and new clinical and laboratory parameters need to be recorded. Searching across numerous tables for all data on a single patient is also inefficient, especially for the vast majority of patients, for whom only a modest number of parameters are actually applicable.

In EAV design, we have (conceptually) a single table that records the data as one row per finding. Each row contains the following information: **entity** (patient identification, visit, date/time etc.), **attribute** (the name/identification of the parameter), and the **value** of the parameter. Because attributes are not hard-coded as database fields, this design does not require revision as new clinical parameters enter the medical domain. Data retrieval is also efficient. To retrieve all the facts on a patient, one simply searches the entity column(s) for the patient identification, ordering all rows by date and time if necessary.

Although EAV architecture dramatically simplifies CDR database design, it complicates user interface design significantly. Specifically, the **global schema** of an EAV database (the way the data are actually organized into tables) differs greatly from its **logical schema** (the way they are perceived as being organized). In our experience, end users tend to regard the data as being stored conventionally as one database field per attribute, even if they are not. Almost all analytic programs, such as spreadsheet or statistics packages, also expect input data to be organized conventionally. Therefore, CDR system architects must spend considerable effort simulating the logical

Affiliation of the authors: Yale University School of Medicine, New Haven, Connecticut.

This work was supported by grant U01-CA-78266 from the National Cancer Institute.

Correspondence and reprints: Prakash M. Nadkarni, MD, Center for Medical Informatics, Yale University School of Medicine, P.O. Box 208009, New Haven, CT 06520-8009;
e-mail: ⟨prakash.nadkarni@yale.edu⟩.

means converting EAV data to a conventional structure before they are presented to the user and translating the edited data back into EAV structure when edits are to be posted back to the server.

There are currently several powerful front-end tools for client–server database development. These include desktop database management systems such as Microsoft Access, Paradox, and Visual Foxpro, as well as programs such as Visual Basic and PowerBuilder. Although such tools often make it possible to create forms without programming, their form design facilities are based on the one-record-per-screen metaphor. They work well for conventional but not for EAV data, where the data on one *logical* record (e.g., all findings pertaining to a single patient event) are stored as multiple *physical* records, with one record per finding. Provision of form interfaces for EAV databases therefore requires much custom programming.

This paper describes WebEAV, a Web-oriented programming framework that minimizes the amount of programming by permitting the automatic generation of Web-based forms for input and display of EAV data. The forms that are generated provide a robust set of features and functionality. We are using WebEAV for two production EAV databases:

- ACT/DB, a database system for managing clinical studies data.[6,7] This is in multidepartment production use at Yale and at the Vanderbilt University Cancer Center. It is also the basis for a special studies database for the U.S. National Cancer Institute–supported Cancer Genetics Network initiative.[8]
- SENSELAB,[9,10] a collection of heterogeneous neuroscience data (sequences, neuronal models, circuits, experiments, etc.) centered on the olfactory system.

Background

Developing Traditional Front Ends for EAV Data

In this section, we discuss the significant maintenance problems that arise when traditional client–server approaches are used to create front ends for complex, multi-user EAV databases. We first describe two approaches for browsing and editing EAV data that are based on mapping sets of attributes to tables that reside on the client and are managed by a desktop database management system. These tables transiently capture data from the server, and the user manipulates the data through forms based on these tables.

ships, where the "many" records are to be displayed simultaneously with the "one" record. For example, a physician inspecting a cancer patient's demographic data may also wish to see details of multiple past episodes of surgery or radiotherapy. Traditional client–server systems handle these presentation needs by letting the developer create *subforms*, one or more of which can be embedded in the main form. Thus, surgery and radiotherapy data, which are also transiently captured in their own tables, are displayed in separate subforms within the demographics form. As discussed later, simulating subforms that also permit data entry and editing on the Web requires complicated programming.

Static Table-based Mapping

In static mapping, each client table reflects an individual data collection instrument. This is a paper-based or electronic form used to gather or present data on a set of related clinical parameters, e.g., routine hematology or a standard clinical chemistry panel such as the SMA-14. Each field on a particular form (e.g., the field "Hemoglobin") is mapped to a counterpart in the EAV schema (the attribute ID for hemoglobin, e.g., 1135). For such purposes, most traditional client–server environments provide a "tag" for every object in a form. The tag can contain arbitrary developer-assigned text whose interpretation is left to program code; thus, the attribute ID can be stored in the tag. The drawbacks of this mapping approach are as follows.

- A large system may require several hundred tables, each with associated forms. The forms and tables can take up considerable space on a client machine, even if the tables are generally empty. It is possible to save space by storing, on clients within individual departments, only those tables and forms that the department needs to use. Maintenance of department-specific sets of tables, however, constitutes significant administrative and manpower overhead.
- Revisions and bug fixes to tables and forms require the corresponding tables or forms to be reinstalled on individual machines. In our experience, many nonstandard data collection instruments that are being devised for brand-new protocols change repeatedly—often four or more times—as investigators iteratively converge on a decision regarding the set of parameters to be gathered.
- The number of form-entry fields per form is limited

data collection instruments (e.g., certain psychiatry questionnaires such as personality inventories). One must then use inelegant workarounds such as splitting up a large data collection instrument into two or more separate forms based on separate tables.

Dynamic Table-based Mapping

Dynamic table-based mapping, a significant improvement over static mapping, was implemented in an earlier version of ACT/DB. Here, the client has a few general-purpose, *reusable* tables with numerous database fields whose mapping to attributes in the EAV schema can change, depending on the form being displayed. That is, these tables are used to transiently capture all EAV, irrespective of the DCI.

One table is used to display data in the main form. The number of subform tables required depends on the maximum number of subforms per form across the system. In our experience, five or six reusable subform tables generally suffice.

The fields in such tables are named serially. In ACT/DB, which uses strong data typing, one designates such fields to hold strings, integers, decimal numbers, dates, and so forth. Thus, the first database field for string data would be named "S01." Strong typing greatly reduces the programming needed for client-side data validation. For example, the built-in validation facilities of many traditional client–server environments prevent entry of alphabets in numeric fields, and date fields reject invalid dates, even handling leap-year logic correctly. "Pictures," which are templates to restrict data entry, such as "(999) 999-9999" for phone numbers, also assist validation and data standardization.

The server's metadata ("data dictionary") records, for every form, the mapping of specific database fields to their corresponding form fields. Subsets of the mapping metadata are replicated programmatically on demand on individual clients, on the basis of the forms that each client uses. When a particular form is about to be opened, its mapping metadata are refreshed from the server if the latter are more recent, as determined by a comparison of time stamps. If the new metadata are incompatible with the old (e.g., the number of fields for one or more data types has changed), the user can be warned that the form on the client is obsolete.

Depending on the client software and setup, it may or may not be possible to automatically download the current version of the form from a "forms server."

have a full version of Access installed. If, however, clients are using Access Runtime (which allows unrestricted application distribution, without per-machine licensing costs), this is not possible, because all forms are treated as having been "compiled" into the application.

By using other metadata—such as the data type and brief description of attributes, the order in which they are to appear in the form, and their aggregation into logical groups—it is possible to write a code library to generate forms, and their mapping metadata, automatically. ACT/DB and SENSELAB both contain such a library, which is described by Nadkarni et al.[11]

Dynamic table-based mapping solves the table proliferation problem, but the maximum-fields-per-form limitation remains. The limit may in fact be reached sooner than with static mapping; for example, all the available string or integer database fields may be used up during the creation of a large form, even if most of the date fields are unused. The dynamic mapping approach also fails to fully address the forms-maintenance problem, because the existence of an obsolete form, while detected correctly, interrupts workflow if the form must be manually downloaded and reinstalled. Furthermore, with a large form the delay caused by metadata downloading and metadata version checking may be significant.

Form Reuse Issues

Multiple departments may use the same data collection instrument with varying degrees of detail. Thus, in a hematology panel, tracking of peripheral promyelocytes or metamyelocytes may be important for cancer chemotherapy but not for routine screening. If one creates numerous department-specific forms for what is fundamentally the same instrument, forms proliferation becomes hard to manage. Reuse of forms that record the greatest common denominator of information is therefore preferred. However, if a given department is concerned with only 5 parameters on a form that has placeholders for 20, it can confuse users who see many more form-entry fields than are appropriate to their needs. Especially if data are being entered through transcription from paper forms, it is important that the data entry person not be presented with fields that do not exist on the paper form.

ACT/DB addresses this issue by permitting the designer to specify, for a given study, which fields on a given form are *required*. When a particular user opens the form, then, based on the current study, the background color of "required" fields is dynamically set

the form is still busier than it should be; ideally, fields that are not required should simply not be shown. One can write generic code to dynamically make non-required fields invisible, but with traditional client–server front ends, the form does not reformat; unsightly gaps indicate invisible fields. Form esthetics becomes a factor if hardcopy is required. Later in this article we discuss how WebEAV addresses this problem.

Creating Web-based Interfaces

The World Wide Web offers a unique opportunity to simplify database deployment. In typical Web database applications, a user's browser requests data from a remote Web server, which in turn requests them from a database server. (The latter may reside on the same machine as the Web server or on a machine in the same network.) After the database server returns the requested data, the Web server formats it into a Web page (in the form of hypertext markup language, or HTML) and sends it to the browser. Additional "application server" software may be placed between Web and database servers; this includes a transaction monitor for tasks such as pooling of database connections to improve response time and reduce database server load.

The advantages of Web deployment are summarized below:

- Problems of maintaining form versions go away, because all forms reside on a Web server, to be downloaded on demand by a client browser. Changes to a given form are automatically available the next time a Web browser accesses the server.
- Web browsers use extremely clever caching algorithms that the developer can leverage. When a browser visits a particular page on a Web site, its contents are cached on the local machine. During a subsequent visit to the same page, only those components (i.e., the HTML, embedded images, applets, or code libraries) that have changed since the last visit are re-downloaded.
- The HTML page- or form-rendering model is both simpler and smarter than that of traditional client–server environments. By default, the objects on a page automatically reformat whenever the browser window is resized or whenever the user changes the font size. Traditional client–server programmers, in contrast, must devote much effort to physical screen size issues. For fine control, "cas-
- the World-Wide Web Consortium (http://www.w3.org/), provide a high-level means of document formatting. A "style" is effectively a macro that permits the font, color, positioning, and visibility of HTML segments to be specified in exquisite detail. Formatting attributes can be altered by "client-side" code. (Client-side code is code that is part of the page and runs in the browser. It is typically written using the language JavaScript or VBScript, or both.) Modest coding efforts achieve dramatic changes in screen appearance.
- Web-based solutions result in significantly lower deployment costs. Browsers are given away free, and therefore per-seat client licenses are not necessary.

For these reasons, the Web is increasingly the medium of choice for multi-user application deployment, especially for databases. However, Web database applications that must support data editing and entry are significantly more complex to develop than traditional client–server applications, for several reasons.

Communication between browser and Web server via the HTTP protocol is intrinsically "stateless."[13] That is, once the server has handed a page to the browser, it closes the connection and "forgets" about the client. To maintain state, the developer must store state data either on a Web page (using "hidden" or invisible form fields) or in "cookies," which are text items in attribute-value form that are stored by the user's browser on the local machine.

Web forms require much more custom programming than traditional client–server environments for client-side data validation, because Web form fields are typeless and "pictures" are unavailable. While server-side validation can be done (and should be done anyway), providing an error message after the Web form is submitted (and after a variable delay) can cause user frustration. Satisfactory ergonomics are facilitated by maximal validation at the client browser *before* form submission, through client-side scripting. Ideally, an error message should appear immediately after the user tries to move from the field with erroneous data to the next field.

In our opinion, many Web development tools are much less mature than traditional client–server environments, and the edit–test–debug cycle is greatly lengthened. Currently, for example, simple errors such as undeclared or misspelled variables, which would be trapped at compile/edit time in traditional client–

need an equivalent of the UNIX *lint* utility,[14] which detects questionable constructs in C code.

Design Objectives

Coding Web forms by hand to support robust data browsing and editing is tedious and error-prone. WebEAV is a framework for simplifying such development. These are its objectives:

- The WebEAV framework should automatically generate forms based on attribute metadata. For efficiency reasons, it is desirable to pregenerate as much of a form as possible, so that most of the form is *static* (i.e., unchanged between consecutive uses). However, the form must also contain *dynamic* components that change the form's behavior on the basis of the currently logged-on user and, in the case of ACT/DB, the current study.
- The esthetics of a program-generated form cannot be 100 per cent satisfactory. It is desirable, therefore, to generate forms that can be customized by (non-programmer) lead users with graphical Web-page editors. Providing form-editing capability enfranchises users and improves user satisfaction while freeing developers for more intellectually challenging tasks.
- The Web forms must be responsive in relatively low-bandwidth situations. Therefore, once a form is downloaded, to-and-fro communication must be minimized. In high-bandwidth traditional client–server applications, in contrast, a client may repeatedly contact the server during data entry, e.g., to populate values in a pull-down menu. A form must therefore contain almost all the scripting code and data, including mapping metadata, needed to function autonomously, until the user submits the form.
- WebEAV should not be limited to managing EAV data alone. Most production EAV databases, including our own, store certain types of homogeneous data, e.g., patient demographics, in conventional form for efficiency purposes.
- The ideas embodied in WebEAV should be sufficiently generic to permit porting to other hardware and software platforms.

Our description of WebEAV in this paper is intended to be comprehensive enough that Web developers in biomedical institutions should be able to derive useful

System Description

WebEAV is currently implemented on the Windows NT platform. It uses Microsoft Internet Information Server as the Web server and Microsoft Transaction Server as the application server; both are part of the default installation of Windows NT Server version 4.0. It uses Active Server Pages, or ASP (described shortly) for server programming. We use Oracle as the database engine (although none of the code in WebEAV is Oracle-specific). On our test system, the database server resides on the same machine as the Web server, while in our production system it resides on a separate machine.

Choice of Software Platform

WebEAV uses ASP technology on the server end. Originally devised by Microsoft for use on their Web server (Internet Information Server, IIS), ASP is also available through a third-party vendor (Chili!Soft) for non-Microsoft Web servers running on non-Windows platforms.

ASP allows a developer to place programming code (written in a "lightweight" scripting language such as VBScript, Javascript, and PerlScript) at multiple places in a Web page. (HTML itself is only a markup language that specifies formatting, not a programming language.) This page is saved with a special file extension (.asp instead of .html). When a browser requests the page, the Web server first passes the page to an interpreter, which executes each instance of embedded code and generates text that is inserted into the page at one or more points. When the browser receives the page, all server-side code has been removed. On Windows NT, the ASP processor and VBScript and JavaScript interpreters are part of the default NT installation.

ASP is not unique in its approach: PHP (http://www.php.net/) is a popular freeware C/Perl-like language environment for UNIX/Windows NT that works on identical principles. Java Server Pages (JSP),[15] which is also available on a variety of Web servers, works in a similar fashion. (The techniques we describe may be readily adapted to PHP or JSP.) The ASP programming model has both advantages and disadvantages.

- ASP offers somewhat higher development throughput than alternatives such as Common Gateway Interface (CGI) programming. (CGI was the first framework defined for Web programming and is

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.