NIH Public Access

Author Manuscript

Int J Med Inform. Author manuscript; available in PMC 2008 November 1.

Published in final edited form as:

Int J Med Inform. 2007 ; 76(11-12): 769–779.

Guidelines for the Effective Use of Entity-Attribute-Value Modeling for Biomedical Databases

Valentin Dinu[a,b] and **Prakash Nadkarni**[a]

[a]*Yale Center for Medical Informatics, New Haven, CT, USA*

[b]*Interdepartmental Program in Computational Biology and Bioinformatics, Yale University, New Haven, CT, USA*

Abstract

Purpose—To introduce the goals of EAV database modeling, to describe the situations where Entity-Attribute-Value (EAV) modeling is a useful alternative to conventional relational methods of database modeling, and to describe the fine points of implementation in production systems.

Methods—We analyze the following circumstances: 1) data are sparse and have a large number of applicable attributes, but only a small fraction will apply to a given entity; 2) numerous classes of data need to be represented, each class has a limited number of attributes, but the number of instances of each class is very small. We also consider situations calling for a mixed approach where both conventional and EAV design are used for appropriate data classes.

Results and Conclusions—In robust production systems, EAV-modeled databases trade a modest data sub-schema for a complex metadata sub-schema. The need to design the metadata effectively makes EAV design potentially more challenging than conventional design.

Keywords

Databases; Entity-Attribute-Value; Clinical Patient Record Systems; Clinical Study Data Management Systems

1. Introduction

Entity-Attribute-Value design is widely used for clinical data repositories (CDRs). The institution/enterprise-level CDRs of Cerner [1] and 3M [2] use an EAV component. EAV, as a general-purpose means of knowledge representation, has its roots in the “association lists” of languages such as LISP, where arbitrary information on any object is recorded as a set of attribute-value pairs [3], and the early object-oriented languages such as SIMULA 67 [4]. The original introduction of EAV design for clinical data storage dates back to the TMR (The Medical Record) system [5] created by Stead and Hammond at Duke in the late 1970s, and the HELP system [6-8]. This model was later given a firm relational-database footing in the Columbia-Presbyterian Medical Center (CPMC) CDR [9-11]. Clinical Study Data Management Systems (CSDMS) that utilize an EAV design include the commercial Phase Forward [12] and Oracle Clinical [13] systems and the open-source TrialDB [14][15,16], developed by our group. The use of EAV design for non-clinical applications is embodied by

Contact for Reprint Requests: Prakash M. Nadkarni, Yale Center for Medical Informatics, PO Box 208009, New Haven, CT 06520-8009, E-mail: Prakash.Nadkarni@yale.edu.

the EAV with classes and relationships (EAV/CR) approach [17,18], as in Yale's SenseLab system [19,20]. The use of EAV/CR is relatively less widespread, and the decision process that determines its appropriate use has not been described previously.

Like any computational approach, EAV design embodies a trade-off. In this case, efficiencies in some aspects are traded off for inefficiencies in others. One therefore needs to understand where this compromise can yield net benefit and the circumstances where it cannot. This paper provides guidelines in the judicious use of EAV design, going beyond simply describing principles, and exploring in depth the infrastructure that must accompany production EAV systems.

In the subsequent text, we will use some standard terms from the object-orientation vocabulary. “Class” means a category of data that has certain properties (descriptors or *attributes*), and an “object” is an instance of a class; “*entity*” is a synonym of “object”. In true object-orientation, classes also have associated subroutines (methods), but methods fall beyond the purview of typical databases, which are concerned with data storage and retrieval, as well as outside the scope of EAV modeling per se.

2. Background

2.1. Attribute-Value Pairs as a General Means of Information Representation

Attribute-value pairs are a venerable method of representing arbitrary information on an object, going back to LISP association lists, which were devised in the 1950s. An example of attribute-value pairs describing a particular car would be: ((VIN XY1876543) (brand Mustang) (color red) (engine V6) (year-of-manufacture 2000)). Most modern programming languages or environments support key-value-pair collections (also called *maps*, *hashes* or *dictionaries*). Maps, however, are memory-based structures that are not intended to address the issue of persistent storage within a database. Relational databases are traditionally designed using at least first normal form (all values are atomic, with no repeating groups), and so attribute-value pairs become *triples* with the entity (the thing being described, identified with a unique identifier of some sort) repeating in each row of a table.

Extensible Markup Language (XML) [21] syntax is related to attribute-value pairs. XML elements, delimited within open- and close-tags for ease and accuracy of parsing, can represent either entities or attributes. They can contain sub-elements nested to arbitrary levels; sub-elements may be regarded as attributes with complex structure. For convenience, atomic data describing an entity may also be represented within an element's open-tag as attribute-value pairs, each component of a pair being separated by an equal sign.

Resource Description Format (RDF) is a standard that is part of the Semantic Web initiative, used to describe Internet-based resources in a standard way. The basic unit of information in RDF is an object-attribute-value triplet [22]. RDF can be represented in XML.

We now consider the circumstances where an attribute-value representation of information is appropriate within a database.

2.2. Limitations of Traditional Database Modeling: Sparseness of Attributes

The conventional way to represent attributes for a class in relational database modeling is as columns in a table, one column per attribute. This approach is suitable for classes with a fixed number of attributes, where most or all attributes have values for a given instance. In a database of cars, for example, it would be natural and appropriate to represent the attributes vehicle identification number (VIN), brand, color, engine type, etc. as individual columns in a Cars table, since all cars should have this information for all attributes.

Columnar attribute representation does not, however, work well for classes with a potentially large number of attributes, where a given instance will have only a few non-empty attributes. (Attribute values may be empty for several reasons, e.g., they are unknown or inapplicable.) In an institutional clinical data repository, the clinical parameters that can apply to a patient across all specialties of medicine number in the hundreds of thousands, but only a relatively small number of parameters are actually recorded for a given patient.

By analogy with the sparse-matrix problem in computer science, we use the term "sparse" for data that show a marked discrepancy between the numbers of potential versus actual attributes. Given that mainstream relational database engines are limited to 1024 columns per table or less, it is not possible to devise a systematic, rational strategy for partitioning the universe of attributes across a hundred or more tables. Further, the parameters continually increase as medical knowledge advances, requiring continual modifications to the schema and to the user interface to the data as well.

2.3. Row modeling for Sparseness and Volatility

To solve the above design problem, consider an analogous situation – the design of a "customer sales" table for a supermarket that has thousands of products in its inventory, where new products are constantly introduced, while others are withdrawn after commercial failure. Product volatility is addressed through a table of Product Information (Universal Product Code (UPC), product name, description, manufacturer, etc.). The basic principle of EAV design is embodied in a sales receipt, which lists only the details of the items that the customer actually purchased, one per row, instead of listing all products that the customer might have purchased but did not. The "entity" – here, the receipt header - records facts such as a transaction number, transaction date/time, store location, etc. Each detail row in the receipt records an "attribute" (here, product) and one or more "values" – e.g., unit price, quantity, discounts and promotions, and total price.

In a database, such information is stored in a "Sales Details" table. This table is said to be *row-modeled*: each *row* records one or more related facts about an entity. Additional sets of facts are recorded through additional rows. Row modeling is a standard component of the database modeler's toolkit, and is not controversial. In the bioinformatics realm, row-modeling is used for genotypic information, where the entity (person/individual) is described by a series of rows, each with an attribute – a genetic locus – and two values corresponding to the detected variants (alleles) for this locus.

Row modeling addresses the twin issues of *sparseness* (the typical customer buys a very small percentage of the available products at any time) and *volatility* (vulnerability of the product list to change). Row modeling should be used when *either* condition exists, but is inappropriate when neither exists: here, traditional (column) modeling should be used.

2.4. EAV Table Design is a Generalization of Row Modeling

Entity-attribute-value design is a generalization of row modeling, where a single table (or set of tables) is used to store all facts affected by sparseness/volatility across the entire database. Table 1 provides an example contrasting EAV and conventional data modeling approaches. The additional circumstances where EAV is preferred over row modeling in special-purpose tables are described below.

- *Individual attributes are heterogeneous with respect to data type*. This is seen with clinical data, with Yes/No items in questionnaires, numeric values in most laboratory parameters, to binary/signal data such as electrocardiographs or images. This makes it difficult (or impossible) to store clinical data for every type of parameter in a single table if a row model is used.

- *Numerous classes of data need to be represented, and their number may continually grow or fluctuate, but the number of instances of each class is very small, even if the attributes for a given class are modest in number and not sparse*. If all classes were represented as individual tables, the database's Entity-Relationship Diagram might have hundreds of classes, with the classes containing thousands/ millions of rows/ instances being emphasized visually to the same extent as classes with very few rows.

  This situation arises in certain kinds of bioscience data (e.g., receptors or ion channels number in the dozens), and may also occur in ontology-modeling environments (where classes must often be created on the fly, and some classes are often eliminated in subsequent cycles of prototyping). For such classes, EAV representation offers a convenient means of persistent storage that leverages the advantage of database technology (such as indexing, views and transactions).

- *Certain (“hybrid”) classes have some attributes that are non-sparse* (present in all or most instances), while other attributes are highly variable and sparse. From the database perspective, the non-sparse attributes are stored conventionally in the table for the class, while the sparse attributes are stored in EAV or row-modeled format. This is seen in the case of description of experiments in microarray databases, where the variable attributes depend on the scientific objective of various categories of experiments – applicable to some categories but not others.

  Hybrid classes are also seen in business database applications. For example, descriptions of products made by a conglomerate corporation depend on the product category, e.g., the attributes necessary to describe a brand of light bulb are quite different from those required to describe a medical imaging device, but both have common attributes such as packaging unit and per-item cost; the system designers may not consider it worthwhile to model each category as a separate class. Note that if only 1-2 classes in the system are hybrid, an EAV design may not be worthwhile. Instead, one would have a special-purpose row-modeled table related many-to-one to the main class table. If numerous classes exhibited hybrid characteristics, then an EAV design would avoid proliferation of these special-purpose tables.

3. Representing the Entity, Value and Attribute

3.1. Representing the Entity

In clinical data, the entity is typically a *Clinical Event*: a logical composite of the patient ID, the date/time when the parameter was measured, optionally with other columns such as the time it was recorded in the database, the protocol in which the patient was enrolled (in the case of clinical studies) and so on. This information is typically captured in a specially designated table whose primary key is a machine-generated number: this number is used as the “Entity” elsewhere.

In bioscience scenarios where one does not have to deal with time-varying data, one way to manage entities is through an “Objects” table that records common information about every “thing” or “object” in the database, typically a preferred name and brief description, as well as the class of entity to which it belongs. A machine-generated primary key value in this table is referenced in other tables as a foreign key, as for clinical data. The “Objects table” approach was pioneered by Tom Slezak and colleagues at Lawrence Livermore Laboratories for the Chromosome 19 database, and is now standard in most large bioinformatics databases, including NCBI Entrez. This approach is orthogonal to EAV modeling, because it does not mandate that the data for individual classes be stored in EAV form: conventional tables can be used if desired. The use of both conventional tables as well as EAV tables in the same schema, with the use of either one or the other as appropriate for a given class of data, is referred to as a *mixed-schema design*. Production schemas are typically mixed.

3.2. Representing Values

Database engines have traditionally required a designer to commit to the data type of a column. Historically (as in the earliest versions of the HELP system) all EAV values were stored using the least-common-denominator data type – i.e., as strings. This approach does not allow effective indexing on the value because indexes on intrinsically numeric or date data stored as strings do not allow optimized range searches, queries specifying comparison operators on numbers must convert the data to numbers on the fly. Alternative approaches include

- Create *multiple value columns* (number, string, date) in a single EAV table with an "indicator" column to identify the column actually containing data for a given row: the other columns will be empty (null). The CPMC CDR employs this design. While space overhead for the table is not significant, index space can be wasted in DBMSs that index null values. (e.g., Microsoft SQL Server and DB/2, but not Oracle).
- *Using separate EAV tables for each data type*. This approach, used in the TrialDB and EAV/CR schemas, involves consulting a given Attribute's definition to determine its data type, and then going to the appropriate table: it is actually quite efficient because the modest amount of metadata for a given set of attributes in a form (or even a study) is typically loaded into memory, using fast structures such as hash tables, prior to accessing the data.
- Microsoft SQL Server allows columns of the "variant" data type: values whose data type is set dynamically and can change as needed. Variants, which are unique to the Microsoft Windows platform, are known to be computationally inefficient. Originally believed to be a convenience for programming in a loosely typed language, in reality their use is highly risky: for example, accidentally treating a particular string variant (e.g., containing the text "ABC") as though it were a number causes a run-time error. Microsoft presently deprecates their use except as a last resort.

Occasionally, during iterative design of a system, the datatype for a particular attribute needs to be reassigned. An example is values that are mostly integer, but which may contain narrative text on occasion. With the first two bulleted approaches, one must move the affected data to a different table or column respectively. This can be done straightforwardly using a stored procedure.

3.2.1. Handling Missing / Approximate Values—In an EAV database, values that are conceptually null – inapplicable, unavailable or unknown – are not generally recorded. Exceptions arise in certain data-entry situations for clinical studies where it is important to record reasons for missing values, e.g., to differentiate between an absent serum potassium due to a hemolyzed blood sample versus a patient's refusal to consent to a blood draw. A generalizable approach for this situation is to add a "missing value code" column to an EAV table, which is non-null only when the value column is null. This code is used to look up a list of study-specific (or system-specific) missing value codes that records the textual explanation for each code.

Another common situation arises when values are known only to a limited precision. This often occurs with dates, which, especially for past history, may be known only to the nearest week, month, year, etc. Here, using an approach originally suggested by Dolin [23], the missing-value-code field can perform double duty, using a set of system-specific codes that indicates the value's precision.

3.2.2. Handling Complex Values—In many situations, notably when EAV is used for bioscience data, values can have sub-structure, as opposed to mostly being atomic for many types of patient data. Specifically, substructure implies that the value is an entity belonging to another class which has its own set of attributes. For example, a neuronal cell has one or more receptor types, where a receptor type is defined by information such as sequence and physiological agonist. The Objects table approach, described earlier, allows us to handle this situation: the value is simply represented as a foreign key that references the objects table. Using the separate-EAV-table approach, the collection of values that happen to be objects is stored in its own EAV table. Retrieval of the details of the sub-structure involves using the value to search the entity column of an EAV table: this may need to be done recursively if sub-structure exists at multiple levels. This is less inefficient than it sounds: for browsing operations that request all details (including details of sub-structure) for a given object, the total number of disk accesses is still very small, and searches on the entity column run fast because it is indexed.

3.3. Representing Attributes

Detailed representation of attributes is the first step toward building a robust metadata infrastructure, a topic discussed in depth later. The list of attributes, typically stored in an “Attributes” table, serves as a local controlled vocabulary: it needs to be curated with the same rigor (e.g., avoidance of duplicate definitions). The primary key of this table, a machine-generated identifier, is used elsewhere.

Several types of metadata are stored for attributes.

- *Validation metadata* include data type, range of permissible values or membership in a set of values, regular expression match, default value, and whether the value is permitted to be null.
- *Presentation metadata* include how the attribute is to be displayed to the user (e.g., as a text box of specified dimensions, a pull-down list or a set of radio buttons): alternative captions that may vary depending on the type of user and/or the language being used (many systems, especially those used for international clinical trials, must have multi-lingual support).
- For attributes which happen to be laboratory parameters, *ranges of normal values*, which may vary by age, sex, physiological state and assay method, are recorded.
- *Grouping metadata*. Attributes are never presented to the user in isolation, but as part of a higher-order group, such as a case report form (CRF)—e.g., a lab test panel. Grouping metadata includes information such as the order in which attributes are presented. Certain presentation metadata, such as fonts/colors and the number of attributes displayed per row (the last is important if the electronic form is to closely resemble a legacy paper form), apply to the group as a whole.
- Other types of metadata include *descriptive metadata,* e.g., detailed descriptions of individual attributes from the perspective of the clinical domain/sub-domain, and possible *mapping to concepts in controlled medical terminologies*. Where such mappings exist, semantic data interchange with other systems becomes possible.

Validation, presentation and grouping metadata make possible the automatic generation of robust user interfaces for both data browsing as well as interactive editing. A framework for using metadata in this way is described in [24].

4. Managing Metadata in an EAV system

An EAV system without a significant metadata component is like an automobile without an engine: it will not function. In an EAV database the physical schema (the way data are stored) is radically different from the logical schema – the way users, and many software applications such as statistics packages, regard it, i.e., as conventional rows and columns for individual classes. The metadata helps perform the sleight-of-hand that lets users interact with the system in terms of the logical schema rather than the physical: the software continually consults the metadata for various operations such as presentation, interactive validation, bulk data extraction and ad hoc query. Metadata that is consulted by software in this way is called "active" metadata: software that is controlled by such metadata is termed "metadata-driven" [25,26]. (The critical dependence on metadata is also seen in other attribute-value-based data representation approaches such as XML, where an XML schema is required to validate a given XML data stream. Similarly, RDF extensions such as the web ontology language (OWL) allow imposing of constraints on the values and formats of data [27].)

EAV systems trade off simplicity in the physical and logical structure of the data for complexity in their metadata. In physical terms, the metadata tables, which are typically represented conventionally because of their homogeneous and non-sparse nature, far outnumber the data tables. The reason for this becomes obvious when we consider below the various roles that metadata plays. Such a tradeoff is generally worthwhile, because in the typical mixed schema of production systems, the data in conventional relational tables can also benefit from functionality such as automatic interface generation.

We now discuss the details of the information that can be captured in metadata.

4.1. Essential Metadata Components: Support of Mixed Schema Designs

At the very least, an EAV system must contain metadata on attributes, as discussed earlier. In a system which records different classes of data, there must similarly be a metadata table for class definitions. In most situations, it is sufficient to model an attribute as belonging to a single class: that is, the Attributes table contains a foreign key that points to the Class Definition table. In the rare circumstance that more than one class happens to share a semantically identical attribute, we would use an alternative design approach, with a "bridge" table recording the many-to-many relationship between classes and attributes.

In a mixed schema design, the class definitions table must have a flag to indicate whether a given class is represented as conventional relational, EAV or hybrid. For a hybrid class, we record against individual attributes how each is represented. This information is essential because it enables the software to present the details of a given object correctly by generating the appropriate data retrieval statements. To ensure that the presented data conform to the logical schema, EAV-structured data need *row-to-columnar transformation* or *pivoting*, a topic important enough to be discussed separately later (in Section 5).

4.2. Metadata for Advanced User Interface Ergonomics and Validation

Metadata can also be used to incorporate application specific logic that can drive the application workflow. We illustrate with examples below.

- <u>Skip Logic:</u> Here, certain attributes/fields on the form are dynamically enabled or disabled based on the values entered in previous attributes. For example, if the response to a question "History of Diabetes?" is "No", then the subsequent questions relating to diabetes severity and therapy are disabled.

Skip-logic metadata tracks dependencies between attributes, as well as the value/s in the controlling attributes that trigger disabling: if the controlling attribute can take multiple values, different attributes may be disabled for each possible value.

- Computed formulas: Here, the values of certain attributes are computed based on the values previously entered in other attributes. A well-known example is the Dubois formula for body surface area, expressed as a function of height and weight. Both skip logic and computed formulas support arbitrary nesting ("chain reaction" behavior), where changes in the value of one attribute can trigger a cascade of changes in others. Computed-formula metadata store the formula itself, as well as inter-attribute dependencies.

  Some practical issues arise here. First, as a system evolves in scope, the formulas that need to be handled by the system get progressively more sophisticated, often calling functions that are created locally and added to a library. Maintaining a parser that can evaluate arbitrarily complex formulas involves significant effort, and the ability to use a "scripting" language (e.g., JavaScript, Perl) that allows dynamic evaluation of expressions, including developer-defined functions, is a much more attractive option. Second, the choice of language determines the syntax used to express the formula. For example, the Dubois formula is expressed mathematically with exponentiation operators: exponentiation is not built into JavaScript and must be supported through an equivalent function.

- Arbitrarily Complex Validation: Here, we have expressions of arbitrary complexity that combine multiple columns; they may also utilize functions that fetch previously stored values from the database; these values are not limited to the parameters on the form, but can access any data in the system, such as patient demographics. The resulting expression is evaluated whenever the user attempts to save a record, and multiple expressions may be evaluated in succession. If a given expression does not evaluate to True, the user receives a corresponding developer-supplied error diagnostic.

  As for computed formulas, the availability of a scripting language that supports evaluation of arbitrary expressions simplifies implementation. We will explore this theme later when we consider how constraints defined in metadata can be implemented in production systems.

- Dynamic pick lists. For fields based on pick lists, the choices presented to the user may occasionally need to change based on previously entered data. An example is seen in forms related to bone marrow or stem cell transplants. A question about the broad category of the disease for which the transplant is being performed (based on a pick list), will lead to a second, "details" question, also based on a pick list, whose choices will change dynamically to be category-specific.

  The metadata to support dynamic pick lists records dependencies between individual items in a "parent" pick list and the "child" list that is to be presented when the user selects the parent item.

Devising a framework that generates metadata-driven user interfaces that support the above features is initially considerably harder than coding on a case-by-case basis. In the long term, however, the costs of application maintenance are considerably reduced.

4.3. Managing Hierarchical/Semantic Relationships between Attributes

In addition to the "grouping" and "dependency" relationships discussed above, semantic relationships also exist between attributes—e.g., certain attributes are associated with particular disease conditions. There are situations where one may wish to query the system to determine what attributes (or higher-order groups of attributes) already exist for a given condition or family of conditions. The most effective way of recording this such information is to utilize a controlled medical terminology, such as UMLS, and use its concepts as “keywords” to tag attributes or attribute groups. The advantage of using controlled terminologies is that one can take advantage of pre-existing concept hierarchies within that terminology, allowing a broader search.

A related situation exists where, within a group of attributes, individual attributes may be treated as different or as identical based on the type of analysis that is performed. For example, within a large institution, different laboratories perform the same lab test (e.g., blood glucose) using different instruments and/or methods, and therefore yielding slightly different numbers for the same sample. In some cases, as when the lab changes its instrumentation, the “normal range” for that parameter may be time-dependent as well.

While this may not be an issue for grossly high/low parameter values, a query of the data asking for minimally elevated or reduced values, if specified with an explicit number, (e.g., 110 mg/dl for blood glucose) might not return meaningful results. This value could be “normal” by one method and “elevated” by another.

There are two computationally equivalent ways to address this situation. One, used in the CPMC CDR, is to treat each lab test variant as a different attribute, and record a hierarchical relationship between these attributes and a “generic” lab-test attribute in metadata. (CPMC stores such hierarchical information in the Columbia Medical Entities Dictionary (MED) [28].) Another approach, used in TrialDB, is to label all variants as the same attribute, but record the details of each variant – e.g., lab, time, ranges of normal – in another table. Both approaches determine elevated/reduced values of the parameter by joining the lab test results to the “normal range” information for each test variant. This approach works even when the normal range is additionally dependent on patient parameters such as age and sex; the join condition, of course, becomes more complicated, and additionally incorporates the patient demographics table.

4.4. Maintaining Data Integrity: Enforcing of Constraints

In a conventional database design, one can use SQL expressions to define constraints (checks) either for individual columns in a table, or define checks in terms of multiple columns. (SQL stands for Structured Query Language, the lingua franca for data definition and manipulation in relational databases.) More elaborate checks, e.g., involving previously entered data, are defined through “triggers”, which are units of code that are automatically activated when a user attempts actions such as creating, modifying or deleting a record. In an EAV system, the constraints on individual attributes and classes are defined in the metadata, but the challenge is to enforce them when the user is creating or modifying data.

EAV databases, like any middle-sized to large-scale database application, are implemented using an “N-tier” approach, where “tier” refers to a layer of software, and “N” refers to a number greater than or equal to two. The “back-end” tier is the database: the “front-end” or “presentation” tier, which interacts with the end-user, is either a desktop or Web browser application. An intervening “middle” layer, typically based on a Web server, performs database access, transaction monitoring and database connection pooling. Technologies such as Java Server Pages or Microsoft ASP.Net are typically deployed at the middle tier.

Any of these tiers can be used to implement constraints: the use of a particular tier has pros and cons, which we now consider.

4.4.1. Enforcement at the Back-End: SQL Constraints and Triggers One possible approach involves automatically translating ("compiling") the constraints defined in metadata into SQL constraints or database triggers: alternatively, one might devise triggers that consulted the metadata dynamically and interpreted the constraints defined therein. The advantage of such an approach, if realized, is that constraints/triggers are impossible to subvert unless they are removed or disabled: they will always activate whenever a data change is attempted, irrespective of the mechanism of attempted change (whether through an administrator tool or an end-user application), or the privilege of the user making the change.

However, there are some practical drawbacks to realizing this approach. Constraints are typically defined in terms of individual columns on a table: this works quite well for conventional one-column-per-attribute design, but not for constraints on an EAV table, where the same column accommodates thousands of attributes with different characteristics, and new attributes are continually added to the system. Therefore interpretation of metadata-defined constraints through triggers is the only viable alternative. The roadblocks to implementing interpretation are described below.

- Triggers contain procedural code. When SQL was originally defined (at a time when the concept of triggers did not exist) it lacked procedural constructs such as branching, loops and subroutines; it was intended to be embedded in a host language that had these features. As a result, when triggers were devised, individual vendors implemented proprietary extensions to the language to enable these capabilities; and triggers were written using these proprietary SQL dialects.

  Such dialects, however, lack the rich library routines and classes (such as hash tables) that characterize modern development environments: some dialects, such as Microsoft/Sybase Transact-SQL, even lack elementary data structures such as arrays. The resulting lack of expressivity limits the complexity of the algorithms that can be expressed in such dialects: for example, writing an expression interpreter in a SQL dialect is often not just difficult but impossible. While ISO SQL-99 now defines "standard" (non-proprietary) procedural constructs for the SQL language, it embodies a case of "too little, too late": it lacks the rich libraries that are a characteristic of modern code development environments such as Java or Microsoft .NET, and which today's programmers have come to expect as a standard part of their armamentarium. Vendor enthusiasm for the procedural extensions of SQL-99 is consequently low to non-existent.

- The more recent versions of Oracle and MS SQL Server 2005 allow trigger code to be written in modern programming languages, e.g., Java, C# or Visual Basic, and provide access to all of the library routines of the underlying environment. While the code development environments for triggers still have their rough edges currently (e.g., in Oracle, debugging facilities are primitive), implementing metadata interpreters may be the way to go in future: tried and tested parser-generator tools such as lex and yacc [29], which now exist in Java form, can be leveraged. However, the challenge of writing an interpreter, particularly one that can handle arbitrary functions created by the developer, must not be under-estimated, and alternative approaches exist that simplify development considerably.

4.4.2. Enforcement in the Middle and Presentation Tiers—In the middle-tier approach, the developer generates Web forms that provide an interface to the data. These forms contain code that performs constraint checking. There are two ways to generate code.

- One can place the expressions stored in the constraints into the code that is part of the Web form: the expressions must follow the syntax of the programming language used for the forms. The Java and Microsoft .NET environments facilitate this approach.

- Some interpreted programming languages used for Web development, notably Perl, JavaScript, and VBScript, support the evaluation of arbitrary text as though it were program code: evaluation succeeds as long as the text conforms to the language syntax. This approach, while possibly less efficient computationally because of the need for dynamic interpretation, can often simplify the programmer's task considerably. The latter two languages can run in the presentation tier and support immediate feedback to the user for data type, range check and regular-expression-pattern-match errors as soon as the user tabs out of a field, without waiting until the user attempts to submit the form. (While JavaScript and VBScript can be used in the middle tier, as in the older Active Server Pages technology, this is not recommended due to the inferior development and debugging environments that their use involves. It is possible, however, to invoke the JavaScript or VBScript engine as needed from a higher-level language.)
- The Web forms can be generated *statically* (in advance) or *dynamically* (on demand). While dynamic generation is potentially impervious to changes in the constraint metadata, our experience is that it is satisfactory only for pure browsing interfaces, where presentation metadata alone needs to be accessed. For implementing the advanced ergonomic and validation features discussed in section 4.2, the delays involved in consulting multiple metadata tables are ergonomically unacceptable on today's server hardware. Static approaches are necessary for forms that must support robust editing: however, manual or automatic mechanisms must be put in place to ensure that the forms are rebuilt whenever the metadata that they depend upon changes. In practice, to minimize the circumstances where forms require rebuilding, a mixture of static and dynamic approaches is used. For example, presentation metadata (captions, choice sets, colors) is relatively inexpensive to access and can be fetched dynamically.

A potential drawback of implementing middle/presentation- tier constraint checking is that someone with system-administrator level privileges can bypass them by going directly to the back-end. Therefore, one must ensure that adequate security policies and mechanisms exist to ensure that data can be modified or created only via the middle tier.

5. Conversion of Row/EAV-Modeled Data to Columnar Data

In order to facilitate working with EAV-modeled data, it is often necessary to transiently or permanently inter-convert between columnar and row-or EAV-modeled representations of the same data[26]. The conversion operation that transforms row-modeled data to columnar representation is called *pivoting*. The circumstances where pivoting is necessary are considered below.

1. *Data browsing of modest amounts of data for an individual entity, optionally followed by data editing based on inter-attribute dependencies*, e.g., skip logic, computed formulas and advanced validation. A typical situation involves inspecting the details of a particular CRF instance for a given patient, followed by edits: alternatively a blank form may be filled in. Such an operation is termed *entity-centric*, and is generally performed on the transactional schema. Here, the data are presented conventionally, with the individual attributes as separate fields. The programming logic that validates the edits treats individual attributes as separate variables, and rules or formulas expressed in terms of these variables operate on a transiently columnar representation of the form's data in memory.
2. Certain *statistical analyses* (e.g., association/correlation analyses) treat individual attributes as separate variables and require bulk-transforming row-modeled data for selected attributes into columnar form. *Data extraction* is the process of performing

bulk transformations of large (but predictable) amounts of data (e.g., a study's complete data) where the results exist as tabular structures to be used until the next extraction.

3. *Ad hoc query interfaces* to row- or EAV-modeled data, when queried from the perspective of individual attributes, (e.g., "retrieve all patients with the presence of liver disease, with either signs of liver failure or the presence of esophageal varices, and no history of alcohol abuse") must typically show the results of the query with individual attributes as separate columns.

The reverse process – conversion of columnar data to EAV data – is seen during the migration of clinical data stored within individual departmental databases into a shared institutional or trans-institutional repository. A limited number of parameters are studied within a specific clinical sub-domain and most of these parameters are recorded for all patients, making a conventional columnar data model appropriate for the departmental databases. Following integration into an institutional repository that deals with multiple clinical specialties, however, the data become sparse, making an EAV model more suitable.

5.1. Transactional Schemas vs. Warehouse Schemas

As the volume of the data being managed in a system gets large, it is generally desirable to maintain a read-only copy of it on a separate machine, which is refreshed from the production (*transactional*) system periodically. The tables in the read-only (*warehouse*) system are typically modified from the transactional system by operations such as denormalization (where several tables are combined into one), and the tables are more extensively indexed. The availability of a warehouse significantly improves the performance of processes 2 and 3 above by avoiding resource contention with a potentially large number of users who are browsing / changing the data.

5.2. Software Strategies for Optimizing Query Performance: Performance Characteristics

From the database perspective, the conceptual operation involved in pivoting is a series of consecutive "full outer joins". Individual "strips" of data, 1 column/attribute wide with a variable number of rows, are joined side by side to create multi-columnar output. The "full outer join" operation accommodates the situation where one of more values of attributes may be missing from individual patients' clinical events: this may occur, for example, due to the operation of skip logic during data entry. Most high-end database vendors support the FULL OUTER JOIN statement in SQL, but in reality, a variety of alternative, but potentially more efficient, approaches can be used that achieve the same end-result. We now consider the different types of output operations where pivoting needs to be performed.

5.2.1. Entity-centric operations—These have been introduced above. Certain entity-centric operations such as, "retrieve *all* details of a particular patient during a specified time period", can run *significantly faster* in EAV, as shown in our group's earlier benchmarking study [30]. This is because a potentially large number of attributes may be scattered across numerous tables in a conventional system, and each table must be accessed sequentially. The EAV schema, by contrast, requires accessing only a few data tables, which are simply filtered by entity and then grouped by whatever attributes are found. The grouping of attributes may be performed based on metadata-defined criteria such as the forms these attributes lie in, the serial order within the form, and chronology (most recent data shown first). Data retrieval is optimized by indexing on patient demographic columns (ID, name), entity (event) IDs, attribute IDs in the EAV tables, and event time stamps. The operation involves:

- Query of the metadata to determine the attributes associated with the given form.
- Identifying the entity or entities by filtering on patient identifiers and time stamps.

- Accessing the appropriate EAV tables for each attribute, filtering the EAV tables by the desired entity/entities.
- Placing the retrieved values in corresponding placeholders in the browsing/editing form.

5.2.2. Ad hoc (attribute-centric) query—Ad hoc query involves retrieving an unpredictable amount of data based on compound Boolean criteria, each of which is based on values of an individual attributes. The operation is termed *attribute-centric*, because we must conceptually extract entity data matching each attribute first, and then combine the intermediate results in Boolean fashion. In the case of the liver disease example above, we could create sets of:

1. All patients where values of a specific serum enzyme, e.g., alanine aminotransferase, exceed a specific threshold.
2. All patients where the finding "alcohol abuse" is explicitly recorded as false (i.e., a significant negative).
3. All patients where the finding "esophageal varices" is true.
4. All patients with a specific liver failure indicator such as elevated levels of blood ammonia.

We would then find patients matching the final desired criterion by performing the compound set operation (Set 1 Intersection Set-2 Intersection (Set-3 Union Set-4).

Attribute-centric queries are important for research purposes, but they do not need to be answered in real time. That is, retrieval time of minutes rather than seconds may be considered acceptable

In our benchmarking study[30], attribute-centric ad hoc queries ran three to twelve times slower in EAV systems, with the slowing being a function of query complexity (number of attributes combined). There are two possible strategies that can be followed here.

1. Generate a series of simple SQL statements, creating temporary tables that are then joined using a tables-with-fewest-rows-first heuristic (for queries that require set intersections) and deleted once the final result table is generated.
2. Generate a giant SQL statement that attempts to render all the desired data in one step.

As shown in [30], the first approach outperforms the second. This is because, despite DBMS engine advances, statements containing numerous joins can create unnecessary work for the database query optimizer [31]. It is not desirable to have the DBMS spend significant time finding the perfect execution plan for a query that will be run just once; all that is needed is a reasonable query plan.

5.2.3. Bulk Data Extraction—The standard bulk extraction operation is "Return complete demographic and clinical data on all patients in a given study." The output for this operation is typically one table for the demographics data, plus one or more relational table per CRF. Multiple tables are created because different CRFs are recorded at different frequencies during the course of the study, as determined by the study protocol. Further, certain CRFs contain unpredictably repeating data. For example, in a CRF recording past episodes of surgery, radiotherapy and chemotherapy, with modality-specific details of each treatment episode, the number of episodes for each treatment modality is highly variable across patients, and the details of each modality go into its own relational table.

This operation can be batched, and does not need to be performed in real time. While a delay of not more than 24 hours is usually acceptable, the algorithms described below can reduce the delay to a few minutes for several gigabytes of data when running on today's off-the-shelf hardware. Bulk extraction has aspects of both entity and attribute-centric query, because we can filter the EAV data based on both a selection of attributes (e.g., those applicable to a given lab panel, or a specific study) and a selection of entities (e.g., all patients in a study). The size of the data set is predictable when there are no complex Boolean criteria: the number of rows, determined by the number of entities, can be ascertained using fairly simple queries, and the number of applicable attributes determines the number of elements per row. In the case of a laboratory test panel, the number of attributes is fixed, while for a clinical study, it varies depending on the individual forms used in the study. In any case, the predictability of output size allows one to opportunistically utilize in-memory data structures to vastly improve extraction performance. There are two approaches for bulk data extraction.

- <u>Extraction in Advance: Materialized Views:</u> A *materialized view* is a table in the read-only warehouse schema that contains the pre-computed results of a SQL SELECT statement, typically one whose results are needed very often, e.g., as the basis for commonly requested reports. In many cases, the SQL statement that creates the view is generated automatically from metadata rather than composed manually, but the choice of approach depends on the desired results: pivoting SQL is simpler to generate. The contents of materialized are refreshed periodically through batch operations on the transactional schema so that their contents are not more than, say, 24 hours out of date, and indexed appropriately for maximal performance. The use of materialized views, which represent a classical space-for-time tradeoff, is not limited to EAV scenarios: in traditional systems, they typically store pre-aggregated summary data, and can dramatically improve reporting performance.

  Materialized views work well in clinical patient record systems (CPRSs) for commonly accessed sets of results such as lab test panels. However, as for any tradeoff, they need to be employed judiciously. The potential number of materialized views can go up non-linearly with the number of higher-order groupings of the data, and the time required to execute the batch operations to create them becomes significant. In a CSDMS that holds dozens of studies, with each study having numerous CRFs, it is not worth materializing every CRF. The same applies to specialty protocols in a CPRS, such as those used for angioplasty or ovarian cancer.

- <u>Extraction on Demand: In-Memory Data Structures:</u> In this approach, one typically operates on denormalized (though not pivoted) EAV data tables in a warehouse to create tabular data that are exported, on demand, to external databases or systems such as statistical packages. It can also be used, however, to create or repopulate materialized views within the warehouse. Our recent work [32] describes the use of hash tables and two-dimensional arrays in memory for pivoting data one CRF at a time and then writing it out to disk, rather than composing giant SQL queries that combine numerous table aliases. This “in-memory” technique significantly outperforms alternative approaches by keeping the queries on EAV tables as simple as possible and minimizing the number of I/O operations: each statement retrieves a large amount of data, and the hash tables help carry out the pivoting operation, which involves placing a value for a given attribute instance into the appropriate row and column. Random Access Memory (RAM) is sufficiently abundant and affordable in modern hardware that the complete data set for most CRFs in even large studies fits into memory. The algorithm, however, performs initial computations to verify that this is the case, and adapts to work on “slices” of the data if it does not fit completely.

6. Conclusions: Judicious Use of the EAV model

As stated earlier, the use of EAV modeling is appropriate for the following situations:

- Data are sparse, heterogeneous, have numerous attributes and new attributes are often needed. This is seen for databases with a clinical data repository component, especially those spanning multiple clinical specialties.
- The number of classes is very large, and numerous classes have a very modest number of instances, even if the attributes for these classes are not necessarily sparse. This is seen in ontology/taxonomy systems and certain bioscience schemas (e.g., neuroscience).
- There are numerous hybrid classes, possessing both sparse and non-sparse attributes. Typically, not all classes in the data will meet the requirements for EAV modeling.

Therefore, production schemas tend to be mixed, with a given class represented in either conventional, EAV or hybrid form as appropriate. The introduction of an EAV component to the schema, however, mandates the creation of a metadata component to capture the logical data model for the EAV data: in the absence of this, the EAV component is essentially unusable. The necessity (and difficulty) of creating a complex meta-schema, as well as a code framework that is driven by it, is one of the major factors that has inhibited the more widespread use of EAV data models: the availability of open-source schemas and frameworks may gradually change this.

Acknowledgements

This work was supported in part by NIH grant U01 CA78266, NIH grant T15 LM07056 from the National Library of Medicine and by institutional funds from Yale University School of Medicine. We thank Prof. Perry Miller of the Yale Center for Medical Informatics for feedback that improved this manuscript.

References

1. Cerner Corporation. The Powerchart Enterprise Clinical Data Repository. 2004 [Sep 9,2004]. web site: http://www.cerner.com/uploadedFiles/1230_03PowerChartFlyer.pdf
2. 3M Health Information Systems. 3M Clinical Data Repository. 2004 [August 8, 2006]. web site: http://www.3m.com/us/healthcare/his/products/records/data_repository.jhtml
3. Steele, G. COMMON LISP: the language. Digital Press; Bedford, Mass: 1990. p. xxiii-1029.
4. Dahl, OJ.; Nygaard, K. Norsk regnesentral., Basic concepts of SIMULA, an ALGOL based simulation language. Norwegian Computing Center; Oslo: 1967. p. 17
5. Stead W, Hammond W, Straube M. A chartless record--is it adequate? Journal of Medical Systems 1983;7:103–109. [PubMed: 6688264]
6. Warner H, Olmsted C, Rutherford B. HELP - a program for medical decision making. Comput Biomed Res 1972;5:65–74. [PubMed: 4553324]
7. Pryor T. The HELP medical record system. MD computing 1988;5:22–33. [PubMed: 3231033]
8. Huff, SM.; Haug, DJ.; Stevens, LE.; Dupont, CC.; Pryor, TA. HELP the next generation: a new client-server architecture. Proc 18th Symposium on Computer Applications in Medical Care; Los Alamitos, CA, Washington, D. C.: IEEE Computer Press; 1994. p. 271-275.
9. Friedman, C.; Hripcsak, G.; Johnson, S.; Cimino, J.; Clayton, P. A Generalized Relational Schema for an Integrated Clinical Patient Database. Proc 14th Symposium on Computer Applications in Medical Care; Los Alamitos, CA, Washington, D. C.: IEEE Computer Press; 1990. p. 335-339.
10. Johnson, S.; Cimino, J.; Friedman, C.; Hripcsak, G.; Clayton, P. Using Metadata to Integrate Medical Knowledge in a Clinical Information System. Proc 14th Symposium on Computer Applications in Medical Care; Los Alamitos, CA, Washington, D. C.: IEEE Computer Press; 1990. p. 340-344.
11. Hripcsak G, Cimino JJ, Sengupta S. WebCIS: large scale deployment of a Web-based clinical information system. Proc AMIA Symp 1999:804–808. [PubMed: 10566471]

12. Phase Forward Inc. ClinTrial. 2004 [10/4/04]. web site: http://www.phaseforward.com/products_cdms_clintrial.html
13. Oracle Corporation. Oracle Clinical Version 3.0: User's Guide. Oracle Corporation; Redwood Shores, CA: 1996.
14. Deshpande A, Brandt C, Nadkarni P. Temporal Query of Attribute-Value Patient Data: Utilizing the Constraints of Clinical Studies. International Journal of Medical Informatics 2003;70:59–77. [PubMed: 12706183]
15. Brandt C, Nadkarni P, Marenco L, Karras B, Lu C, Schacter L, Fisk J, PL M. Reengineering a database for clinical trials management: lessons for system architects. Controlled Clinical Trials 2000;21:440–461. [PubMed: 11018562]
16. Nadkarni PM, Brandt C, Frawley S, Sayward F, Einbinder R, Zelterman D, Schacter L, Miller PL. Managing attribute-value clinical trials data using the ACT/DB client-server database system. Journal of the American Medical Informatics Association 1998;5:139–151. [PubMed: 9524347]
17. Nadkarni PM, Marenco L, Chen R, Skoufos E, Shepherd G, Miller P. Organization of Heterogeneous Scientific Data Using the EAV/CR Representation. Journal of the American Medical Informatics Association 1999;6:478–493. [PubMed: 10579606]
18. Marenco L, Tosches N, Crasto C, Shepherd G, Miller P, Nadkarni P. Achieving evolvable Web-database bioscience applications using the EAV/CR framework: recent advances. J Am Med Inform Assoc 2003;10:444–453. [PubMed: 12807806]
19. Shepherd, GM.; Healy, MD.; Singer, MS.; Peterson, BE.; Mirsky, JS.; Wright, L.; Smith, JE.; Nadkarni, PM.; Miller, PL. Senselab: a project in multidisciplinary, multilevel sensory integration. In: Koslow, SH.; Huerta, MF., editors. Neuroinformatics: An Overview of the Human Brain Project. Lawrence Erlbaum Associates, Inc.; Mahwah, NJ: 1997. p. 21-56.
20. Marenco L, Nadkarni P, Skoufos E, Shepherd G, Miller P. Neuronal database integration: the Senselab EAV data model. Proceedings of the AMIA Symposium 1999:102–106.
21. W.W.W. Consortium. Extensible Markup Language (XML) 1.0, W3C Recommendation. 2004 [August 8, 2006]. web site: http://www.w3.org/TR/REC-xml/
22. Introduction to Semantic Web Technologies. [February 5, 2006]. web site: http://www.hpl.hp.com/semweb/sw-technology.htm
23. Dolin RH. Modelling the temporal complexities of symptoms. JAMIA 1995;2:323–331. [PubMed: 7496882]
24. Nadkarni PM, Brandt CA, Marenco L. WebEAV: Automatic Metadata-driven Generation of Web Interfaces to Entity-Attribute-Value Databases. Journal of the American Medical Informatics Association 2000;7:343–356. [PubMed: 10887163]
25. Vaduva A, Vetterli T. Metadata Management for Data Warehousing: An Overview. International Journal of Cooperative Information Systems 2001;10:273–298.
26. Brandt C, Morse R, Matthews K, Sun K, Deshpande A, Gadagkar R, Cohen D, PL M, Nadkarni P. Metadata-driven Creation of Data Marts from an EAV-Modeled Clinical Research Database. International Journal of Medical Informatics 2002;65:225–241. [PubMed: 12414020]
27. Web Ontology Language (OWL). [2/16/2006]. web site: http://www.w3.org/TR/owl-ref/
28. Cimino JJ, Clayton PD, Hripcsak G, Johnson SB. Knowledge-based approaches to the maintenance of a large controlled medical terminology. JAMIA 1994;1:35–50. [PubMed: 7719786]
29. Levine, JR.; Mason, T.; Brown, D. lex & yacc. O'Reilly & Associates, Inc.; Sebastopol, CA: 1992.
30. Chen RS, Nadkarni PM, Marenco L, Levin FW, Erdos J, Miller PL. Exploring Performance Issues for a Clinical Database Organized Using an Entity-Attribute-Value Representation. Journal of the American Medical Informatics Association 2000;7:475–487. [PubMed: 10984467]
31. Nadkarni P, Brandt C. Data Extraction and Ad Hoc Query of an Entity-Attribute-Value Database. Journal of the American Medical Informatics Association 1998;5:511–527. [PubMed: 9824799]
32. Dinu V, Nadkarni P, Brandt C. Pivoting approaches for bulk extraction of Entity-Attribute-Value data. Comput Methods Programs Biomed 2006;82:38–43. [PubMed: 16556470]

Table 1

Comparison of how the same data is stored in a conventional relational database model (A) and an EAV database model (B), using a simplified data set. There is no need to create entries for not available ("N/A") data in the EAV model. Written in bold fonts, the addition of a new attribute, Test3, for the patient with Id 3, requires the modification of the database schema through the addition of a new column in the conventional table, while in the EAV model it only requires the creation of a new row. In practice, instead of using a single EAV table, we use multiple tables, based on the data type of individual attributes.

Most analytical programs expect their input data to be in columnar format rather than EAV structure. The pivoting operation discussed in Section 5 transforms the EAV-modeled data from Table 1B into a columnar format like that from Table 1A.

Table 1A. Clinical data stored using a conventional relational database model

Patient Id	Date	Test1	Test2	Test3
1	6/15/2004	TRUE	100	**N/A**
1	1/30/2005	TRUE	110	**N/A**
2	3/10/2003	FALSE	N/A	**N/A**
3	2/15/2005	TRUE	90	**50**

Table 1B. Clinical data stored using an EAV database model.

Entity	Attribute	Value
1:6/15/2004	Test1	TRUE
1:6/15/2004	Test2	100
1:1/30/2005	Test1	TRUE
1:1/30/2005	Test2	110
2:3/10/2003	Test1	FALSE
3:2/15/2005	Test1	TRUE
3:2/15/2005	Test2	90
3:2/15/2005	**Test3**	**50**