Entity–attribute–value model

From Wikipedia, the free encyclopedia

Entity–attribute–value model (**EAV**) is a data model to describe entities where the number of attributes (properties, parameters) that can be used to describe them is potentially vast, but the number that will actually apply to a given entity is relatively modest. In mathematics, this model is known as a sparse matrix. EAV is also known as **object–attribute–value model**, **vertical database model** and **open schema**.

Contents

Structure of an EAV table

This data representation is analogous to space-efficient methods of storing a sparse matrix, where only non-empty values are stored. In an EAV data model, each attribute-value pair is a fact describing an entity, and a row in an EAV table stores a single fact. EAV tables are often described as "long and skinny": "long" refers to the number of rows, "skinny" to the few columns.

Data is recorded as three columns:

- The *entity*: the item being described.
- The *attribute* or *parameter*: a foreign key into a table of attribute definitions. At the very least, the attribute definitions table would contain the following columns: an attribute ID, attribute name, description, data type, and columns assisting input validation, e.g., maximum string length and regular expression, set of permissible values, etc.
- The *value* of the attribute.

Consider how one would try to represent a general-purpose clinical record in a relational database. Clearly creating a table (or a set of tables) with thousands of columns is not the way to go, because the vast majority of columns would be null. To complicate things, in a longitudinal medical record that follows the patient over time, there may be multiple values of the same parameter: the height and weight of a child, for example, change as the child grows. Finally, the universe of clinical findings keeps growing: for example, diseases such as SARS emerge, and new lab tests are devised; this would require constant addition of columns, and constant revision of the user interface. (The situation where the list of attributes changes frequently is termed "attribute volatility" in database parlance.)

The following shows a snapshot of an EAV table for clinical findings. The entries shown within angle brackets are references to entries in other tables, shown here as text rather than as encoded foreign key values for ease of understanding. They represent some details of a visit to a doctor for fever on the morning of 1/5/98. In this example, the **values** are all literal values, but these could also be foreign keys to pre-defined value lists; these are particularly useful when the possible values are known to be limited.

- The *entity*. For clinical findings, the entity is the *patient event*: a foreign key into a table that contains at a minimum a patient ID and one or more time-stamps (e.g., the start and end of the examination date/time) that record when the event being described happened.
- The *attribute* or *parameter*: a foreign key into a table of attribute definitions (in this example, definitions of clinical findings). At the very least, the attribute definitions table would contain the following columns: an attribute ID, attribute name, description, data type, units of measurement, and columns assisting input validation, e.g., maximum string length and regular expression, maximum and minimum permissible values, set of permissible values, etc.
- The *value* of the attribute. This would depend on the data type, and we discuss how values are stored shortly.

The example below illustrates symptoms findings that might be seen in a patient with pneumonia.

```
(<patient XYZ, 1/5/98 9:30 AM>,  <Temperature in degrees Fahrenheit>,  "102" )

(<patient XYZ, 1/5/98 9:30 AM>,  <Presence of Cough>,  "True" )

(<patient XYZ, 1/5/98 9:30 AM>,  <Type of Cough>,  "With phlegm, yellowish, streaks of blood" )

(<patient XYZ, 1/5/98 9:30 AM>,  <Heart Rate in beats per minute>,  "98" )

...
```

EAV databases

The term "EAV database" refers to a database design where a significant proportion of the data is modeled as EAV. However, even in a database described as "EAV-based", some tables in the system are traditional relational tables.

- As noted above, EAV modeling makes sense for categories of data, such as clinical findings, where attributes are numerous and sparse. Where these conditions do not hold, standard relational modeling (i.e., one column per attribute) is preferable; using EAV does not mean abandoning common sense or principles of good relational design. In clinical record systems, the subschemas dealing with patient demographics and billing are typically modeled conventionally. (While most vendor database schemas are proprietary, VistA, the system used throughout the United States Department of Veterans Affairs (VA) medical system, known as the Veterans Health Administration (VHA),[1] is open-source and its schema is readily inspectable, though it uses a MUMPS database engine rather than a relational database.)

- As discussed shortly, an EAV database is essentially unmaintainable without numerous supporting tables that contain supporting metadata. The metadata tables, which typically outnumber the EAV tables by a factor of at least three or more, are typically standard relational tables.[2][3] An example of a metadata table is the Attribute Definitions table mentioned above.

EAV versus row modeling

The EAV data described above is comparable to the contents of a supermarket sales receipt (which would be reflected in a Sales Line Items table in a database). The receipt lists only details of the items actually purchased, instead of listing every product in the store that the customer might have purchased but didn't. Like the clinical findings for a given patient, the sales receipt is sparse.

- The "entity" is the sale/transaction id — a foreign key into a sales transactions table. This is used to tag each line item internally, though on the receipt the information about the Sale appears at the top (store location, sale date/time) and at the bottom (total value of sale).
- The "attribute" is a foreign key into a products table, from where one looks up description, unit price, discounts and promotions, etc. (Products are just as volatile as clinical findings, possibly even more so: new products are introduced every month, while others are taken off the market if consumer acceptance is poor. No competent database designer would hard-code individual products such as Doritos or Diet Coke as columns in a table.)
- The "values" are the quantity purchased and total line item price.

Row modeling, where facts about something (in this case, a sales transaction) are recorded as multiple *rows* rather than multiple *columns*, is a standard data modeling technique. The differences between row modeling and EAV (which may be considered a *generalization* of row-modeling) are:

- A row-modeled table is *homogeneous* in the facts that it describes: a Line Items table describes only products sold. By contrast, an EAV table contains almost any type of fact.
- The data type of the value column/s in a row-modeled table is pre-determined by the nature of the facts it records. By contrast, in an EAV table, the conceptual data type of a value in a particular row depend on the attribute in that row. It follows that in production systems, allowing direct data entry into an EAV table would be a recipe for disaster, because the database engine itself would not be able to perform robust input validation. We shall see later how it is possible to build generic frameworks that perform most of the tasks of input validation, without endless coding on an attribute-by-attribute basis.

In a clinical data repository, row modeling also finds numerous uses; the laboratory test subschema is typically modeled this way, because lab test results are typically numeric, or can be encoded numerically.

The circumstances where you would need to go beyond standard row-modeling to EAV are listed below:

- The data type of individual attributes varies (as seen with clinical findings).
- The categories of data are numerous, growing or fluctuating, but the number of instances (records/rows) within each category is very small. Here, with conventional modeling, the database's entity–relationship diagram might have hundreds of tables: the tables that contain

thousands/ millions of rows/instances are emphasized visually to the same extent as those with very few rows. The latter are candidates for conversion to an EAV representation.

This situation arises in ontology-modeling environments, where categories ("classes") must often be created on the fly, and some classes are often eliminated in subsequent cycles of prototyping.

- Certain ("hybrid") classes have some attributes that are non-sparse (present in all or most instances), while other attributes are highly variable and sparse. The latter are suitable for EAV modeling. 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.

Physical representation of EAV data

The Entity

In clinical data, the entity is typically a clinical event, as described above. In more general-purpose settings, the entity is a foreign key into an "objects" table that records common information about every "object" (thing) in the database – at the minimum, a preferred name and brief description, as well as the category/class of entity to which it belongs. Every record (object) in this table is assigned a machine-generated object ID.

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. The use of an objects table does not mandate the concurrent use of an EAV design: conventional tables can be used to store the category-specific details of each object.

The major benefit to a central objects table is that, by having a supporting table of object synonyms and keywords, one can provide a standard Google-like search mechanism across the entire system where the user can find information about any object of interest without having to first specify the category that it belongs to. (This is important in bioscience systems where a keyword like "acetylcholine" could refer either to the molecule itself, which is a neurotransmitter, or the biological receptor to which it binds.

The Attribute

In the EAV table itself, this is just an attribute ID, a foreign key into an Attribute Definitions table, as stated above. However, there are usually multiple metadata tables that contain attribute-related information, and these are discussed shortly.

The Value

Coercing all values into strings, as in the EAV data example above, results in a simple, but non-scalable, structure: constant data type inter-conversions are required if one wants to do anything with the values, and an index on the value column of an EAV table is essentially useless. Also, it is not convenient to store large binary data, such as images, in Base64 encoded form in the same table as small integers or strings. Therefore larger systems use separate EAV tables for each data type (including binary large objects, "BLOBS"), with the metadata for a given attribute identifying the EAV table in which its data will be stored. This approach is actually quite efficient because the modest amount of attribute metadata for a given class or form that a user chooses to work with can be cached readily in memory. However, it requires moving of data from one table to another if an attribute's data type is changed. (This does not happen often, but mistakes can be made in metadata definition just as in database schema design.)

Representing substructure: EAV with classes and relationships (EAV/CR)

In a simple EAV design, the values of an attribute are simple or primitive data types as far as the database engine is concerned. However, in EAV systems used for representation of highly diverse data, it is possible that a given object (class instance) may have substructure: that is, some of its attributes may represent other kinds of objects, which in turn may have substructure, to an arbitrary level of complexity. A car, for example, has an engine, a transmission, etc., and the engine has components such as cylinders. (The permissible substructure for a given class is defined within the system's attribute metadata, as discussed later. Thus, for example, the attribute "random-access-memory" could apply to the class "computer" but not to the class "engine".)

To represent substructure, one incorporates a special EAV table where the value column contains references to *other* entities in the system (i.e., foreign key values into the objects table). To get all the information on a given object requires a recursive traversal of the metadata, followed by a recursive traversal of the data that stops when every attribute retrieved is simple (atomic). Recursive traversal is necessary whether details of an individual class are represented in conventional or EAV form; such traversal is performed in standard object–relational systems, for example. In practice, the number of levels of recursion tends to be relatively modest for most classes, so the performance penalties due to recursion are modest.

EAV/CR (EAV with Classes and Relationships) refers to a framework that supports complex substructure. Its name is somewhat of a misnomer: while it was an outshoot of work on EAV systems, in practice, many or even most of the classes in such a system may be represented in standard relational form, based on whether the attributes are sparse or dense. EAV/CR is really characterized by its very detailed metadata, which is rich enough to support the automatic generation of browsing interfaces to individual classes without having to write class-by-class user-interface code.

The critical role of metadata in EAV systems

In the words of Prof. Dr. Daniel Masys (formerly Chair of Vanderbilt University's Medical Informatics Department), the challenges of working with EAV stem from the fact that 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. (Because an EAV table conceptually mixes apples, oranges, grapefruit and chop suey, if you want to do any analysis of the data using standard off-the-shelf software, in most cases you have to convert subsets of it into columnar form.[4] The process of doing this, called pivoting, is important enough to be discussed separately.)

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 data presentation, interactive validation, bulk data extraction and ad hoc query. The metadata can actually be used to customize the behavior of the system.

EAV systems trade off simplicity in the physical and logical structure of the data for complexity in their metadata, which, among other things, plays the role that database constraints and referential integrity do in standard database designs. 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. The structure of the metadata is complex enough that it comprises its own subschema within the database: various foreign keys in the data tables refer to tables within this subschema. This subschema is standard-relational, with features such as constraints and referential integrity being used to the hilt.

The correctness of the metadata contents, in terms of the intended system behavior, is critical and the task of ensuring correctness means that, when creating an EAV system, considerable design efforts must go into building user interfaces for metadata editing that can be used by people on the team who know the problem domain (e.g., clinical medicine) but are not necessarily programmers. (Historically, one of the main reasons why the pre-relational TMR system failed to be adopted at sites other than its home institution was that all metadata was stored in a single file with a non-intuitive structure. Customizing system behavior by altering the contents of this file, without causing the system to break, was such a delicate task that the system's authors only trusted themselves to do it.)

Where an EAV system is implemented through RDF, the RDF Schema language may conveniently be used to express such metadata. This Schema information may then be used by the EAV database engine to dynamically re-organize its internal table structure for best efficiency.[5]

Some final caveats regarding metadata:

- Because the business logic is in the metadata rather than explicit in the database schema (i.e., one level removed, compared with traditionally designed systems), it is less apparent to one who is unfamiliar with the system. Metadata-browsing and metadata-reporting tools are therefore important in ensuring the maintainability of an EAV system. In the common scenario where metadata is implemented as a relational sub-schema, these tools are nothing more than applications built using off-the-shelf reporting or querying tools that operate on the metadata tables.
- It is easy for an insufficiently knowledgeable user to corrupt (i.e., introduce inconsistencies and errors in) metadata. Therefore, access to metadata must be restricted, and an audit trail of accesses and changes put into place to deal with situations where multiple individuals have metadata access. Using an RDBMS for metadata will simplify the process of maintaining consistency during metadata creation and editing, by leveraging RDBMS features such as support for transactions. Also, if the metadata is part of the same database as the data itself, this ensures that it will be backed up at least as frequently as the data itself, so that it can be recovered to a point in time.
- The quality of the annotation and documentation within the metadata (i.e., the narrative/explanatory text in the descriptive columns of the metadata sub-schema) must be much higher, in order to facilitate understanding by various members of the development team. Ensuring metadata quality (and keeping it current as the system evolves) takes very high priority in the long-term management and maintenance of any design that uses an EAV component. Poorly-documented or out-of-date metadata can compromise the system's long-term viability [1] (http://www.amazon.com/dp/0857295098).[6]

Information captured in metadata

Attribute metadata

- **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. In EAV systems representing classes with substructure, the validation metadata will also record what class, if any, a given attribute belongs to.

- **Presentation metadata**: how the attribute is to be displayed to the user (e.g., as a text box or image of specified dimensions, a pull-down list or a set of radio buttons).
- 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 typically presented as part of a higher-order group, e.g., a specialty-specific form. 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, apply to the group as a whole.

Advanced validation metadata

- **Dependency metadata**: in many user interfaces, entry of specific values into certain fields/attributes is required to either disable/hide certain other fields or enable/show other fields. (For example, if a user chooses the response "No" to a Boolean question "Does the patient have diabetes?", then subsequent questions about the duration of diabetes, medications for diabetes, etc. must be disabled.) To effect this in a generic framework involves storing of dependencies between the controlling attributes and the controlled attributes.
- **Computations and complex validation**: As in a spreadsheet, the value of certain attributes can be computed, and displayed, based on values entered into fields that are presented earlier in sequence. (For example, body surface area is a function of height and width). Similarly, there may be "constraints" that must be true for the data to be valid: for example, in a differential white cell count, the sum of the counts of the individual white cell types must always equal 100, because the individual counts represent percentages. Computed formulas and complex validation are generally effected by storing expressions in the metadata that are macro-substituted with the values that the user enters and can be evaluated. In Web browsers, both JavaScript and VBScript have an Eval() function that can be leveraged for this purpose.

Validation, presentation and grouping metadata make possible the creation of code frameworks that support automatic user interface generation for both data browsing as well as interactive editing. In a production system that is delivered over the Web, the task of validation of EAV data is essentially moved from the back-end/database tier (which is powerless with respect to this task) to the middle /Web server tier. While back-end validation is always ideal, because it is impossible to subvert by attempting direct data entry into a table, middle tier validation through a generic framework is quite workable, though a significant amount of software design effort must go into building the framework first. The availability of open-source frameworks that can be studied and modified for individual needs can go a long way in avoiding wheel reinvention.

Scenarios that are appropriate for EAV modeling

(The first part of this section is a precis of the Dinu/Nadkarni reference article in PubMed Central,[7] to which the reader is directed for more details.)

EAV modeling, under the alternative terms "generic data modeling" or "open schema", has long been a standard tool for advanced data modelers. Like any advanced technique, it can be double-edged, and should be used judiciously.

Also, the employment of EAV does not preclude the employment of traditional relational database modeling approaches within the same database schema. In EMRs that rely on an RBDMS, such as Cerner, which use an EAV approach for their clinical-data subschema, the vast majority of tables in the schema are in fact traditionally modeled, with attributes represented as individual columns rather than as rows.

The modeling of the metadata subschema of an EAV system, in fact, is a very good fit for traditional modeling, because of the inter-relationships between the various components of the metadata. In the TrialDB system, for example, the number of metadata tables in the schema outnumber the data tables by about ten to one. Because the correctness and consistency of metadata is critical to the correct operation of an EAV system, the system designer wants to take full advantages of all of the features that RDBMSs provide, such as referential integrity and programmable constraints, rather than having to reinvent the RDBMS-engine wheel. Consequently, the numerous metadata tables that support EAV designs are typically in third-normal relational form.

Modeling sparse attributes

The typical case for using the EAV model is for highly sparse, heterogeneous attributes, such as clinical parameters in the electronic medical record (EMRs), as stated above. Even here, however, it is accurate to state that the EAV modeling principle is applied to a *sub-schema* of the database rather than for all of its contents. (Patient demographics, for example, are most naturally modeled in one-column-per-attribute, traditional relational structure.)

Consequently, the arguments about EAV vs. "relational" design reflect incomplete understanding of the problem: An EAV design should be employed only for that sub-schema of a database where sparse attributes need to be modeled: even here, they need to be supported by third normal form metadata tables. There are relatively few database-design problems where sparse attributes are encountered: this is why the circumstances where EAV design is applicable are relatively rare. Even where they are encountered, a set of EAV tables is not the only way to address sparse data: an XML-based solution (discussed below) is applicable when the maximum number of attributes per entity is relatively modest, and the total volume of sparse data is also similarly modest. An example of this situation is the problems of capturing variable attributes for different product types.

Modeling numerous classes with very few instances per class: highly dynamic schemas

Another application of EAV is in modeling classes and attributes that, while not sparse, are dynamic, but where the number of data rows per class will be relatively modest – a couple of hundred rows at most, but typically a few dozen – and the system developer is also required to provide a Web-based end-user interface within a very short turnaround time. "Dynamic" means that new classes and attributes need to be continually defined and altered to represent an evolving data model. This scenario can occur in rapidly evolving scientific fields as well as in ontology development, especially during the prototyping and iterative refinement phases.

While creation of new tables and columns to represent a new category of data is not especially labor-intensive, the programming of Web-based interfaces that support browsing or basic editing with type- and range-based validation is. In such a case, a more maintainable long-term solution is to create a framework where the class and attribute definitions are stored in metadata, and the software generates a basic user interface from this metadata dynamically.

The EAV/CR framework, mentioned earlier, was created to address this very situation. Note that an EAV data model is not essential here, but the system designer may consider it an acceptable alternative to creating, say, sixty or more tables containing a total of not more than two thousand rows. Here, because the number of rows per class is so few, efficiency considerations are less important; with the standard indexing by class ID/attribute ID, DBMS optimizers can easily cache the data for a small class in memory when running a query involving that class or attribute.

In the dynamic-attribute scenario, it is worth noting that Resource Description Framework (RDF) is being employed as the underpinning of Semantic-Web-related ontology work. RDF, intended to be a general method of representing information, is a form of EAV: an RDF triple comprises an object, a property, and a value.

At the end of Jon Bentley's book "Writing Efficient Programs", the author warns that making code more efficient generally also makes it harder to understand and maintain, and so one does not rush in and tweak code unless one has first determined that there **is** a performance problem, and measures such as code profiling have pinpointed the exact location of the bottleneck. Once you have done so, you modify only the specific code that needs to run faster. Similar considerations apply to EAV modeling: you apply it only to the sub-system where traditional relational modeling is known *a priori* to be unwieldy (as in the clinical data domain), or is discovered, during system evolution, to pose significant maintenance challenges.

Working with EAV data

The Achilles heel of EAV is the difficulty of working with large volumes of EAV data. It is often necessary to transiently or permanently inter-convert between columnar and row-or EAV-modeled representations of the same data; this can be both error-prone if done manually as well as CPU-intensive. Generic frameworks that utilize attribute and attribute-grouping metadata address the former but not the latter limitation; their use is more or less mandated in the case of mixed schemas that contain a mixture of conventional-relational and EAV data, where the error quotient can be very significant.

The conversion operation is called **pivoting**. Pivoting is not required only for EAV data but also for any form or row-modeled data. (For example, implementations of the Apriori algorithm for Association Analysis, widely used to process supermarket sales data to identify other products that purchasers of a given product are also likely to buy, pivot row-modeled data as a first step.) Many database engines have proprietary SQL extensions to facilitate pivoting, and packages such as Microsoft Excel also support it. The circumstances where pivoting is necessary are considered below.

- **Browsing** of modest amounts of data for an individual entity, optionally followed by data editing based on inter-attribute dependencies. This operation is facilitated by caching the modest amounts of the requisite supporting metadata. Some programs, such as TrialDB, access the metadata to generate semi-static Web pages that contain embedded programming code as well as data structures holding metadata.
- **Bulk extraction** transforms large (but predictable) amounts of data (e.g., a clinical study's complete data) into a set of relational tables. While CPU-intensive, this task is infrequent and does not need to be done in real-time; i.e., the user can wait for a batched process to complete. The importance of bulk extraction cannot be overestimated, especially when the data is to be processed or analyzed with standard third-party tools that are completely unaware of EAV structure. Here, it is not advisable to try to reinvent entire sets of wheels through a generic framework, and it is best just to bulk-extract EAV data into relational tables and then work with it using standard tools.
- **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 signs of liver failure and no history of alcohol abuse") must typically show the results of the query with individual attributes as separate columns. For most EAV database scenarios ad hoc query performance must be tolerable, but sub-second responses are not necessary, since the queries tend to be exploratory in nature.

Relational division

However, the structure of EAV data model is a perfect candidate for Relational Division, see relational algebra. With a good indexing strategy it's possible to get a response time in less than a few hundred milliseconds on a billion row EAV table. Microsoft SQL Server MVP Peter Larsson has proved this on a laptop and made the solution general available.

Optimizing pivoting performance

- One possible optimization is the use of a separate "**warehouse**" or queryable schema whose contents are refreshed in batch mode from the production (transaction) schema. See data warehousing. The tables in the warehouse are heavily indexed and optimized using denormalization, which combines multiple tables into one to minimize performance penalty due to table joins. This is the approach that Kalido uses to convert highly normalized EAV tables to standard reporting schemas.
- Certain EAV data in a warehouse may be converted into standard tables using "**materialized views**" (see data warehouse), but this is generally a last resort that must be used carefully, because the number of views of this kind tends to grow non-linearly with the number of attributes in a system.[4]
- **In-memory data structures**: One can use hash tables and two-dimensional arrays in memory in conjunction with attribute-grouping metadata to pivot data, one group at a time. This data is written to disk as a flat delimited file, with the internal names for each attribute in the first row: this format can be readily bulk-imported into a relational table. 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.[4] 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 a single attribute group in even large data sets will usually fit completely into memory, though the algorithm can be made smarter by working on slices of the data if this turns out not to be the case.

Obviously, no matter what approaches you take, querying EAV will never be as fast as querying standard column-modeled relational data, in much the same way that access of elements in sparse matrices are not as fast as those on non-sparse matrices if the latter fit entirely into main memory. (Sparse matrices, represented using structures such as linked lists, require list traversal to access an element at a given X-Y position, while access to elements in matrices represented as 2-D arrays can be performed using fast CPU register operations.) If, however, you chose the EAV approach correctly for the problem that you were trying to solve, this is the price that you pay; in this respect, EAV modeling is an example of a space (and schema maintenance) versus CPU-time tradeoff.

Consideration for SQL Server 2008 and later: Sparse columns

Microsoft SQL Server 2008 offers a (proprietary) alternative to EAV: columns with an atomic data type (e.g., numeric, varchar or datetime columns) can be designated as *sparse* simply by including the word SPARSE in the column definition of the CREATE TABLE statement. Sparse columns optimize the storage of NULL values (which now take up no space at all) and are useful when the majority records in a table will have NULL values for that column. Indexes on sparse columns are also optimized: only those rows with values are indexed. In addition, the contents of all sparse columns in a particular row of a table can be collectively aggregated into a single XML column (a column set), whose contents are of the form `[<column-name>column contents </column-name>]*....` In fact, if a column set is defined for a table as part of a CREATE TABLE statement, all sparse columns subsequently defined are typically added to it. This has the interesting consequence that the SQL statement `SELECT * from <tablename>` will not return the individual sparse columns, but concatenate all of them into a single XML column whose name is that of the column set (which therefore acts as a virtual, computed column).

Sparse columns are convenient for business applications such as product information, where the applicable attributes can be highly variable depending on the product type, but where the total number of variable attributes per product type are relatively modest. Limitations include:

- The maximum number of sparse columns in a table is 30,000, which may fall short for some implementations, such as for storing clinical data, where the possible number of attributes is one order of magnitude larger. Therefore, this is not a solution for modeling *all* possible clinical attributes for a patient.
- Addition of new attributes – one of the primary reasons an EAV model might be sought – still requires a DBA, and the problem of building a user interface to such data is not addressed: only the storage mechanism is streamlined. Applications can be written to dynamically add and remove sparse columns from a table at run-time: in contrast an attempt to perform such an action in a multi-user scenario where other users/processes are still using the table would be prevented for tables without sparse columns. This capability offers power and flexibility, but can result in significant performance penalties, in part because any compiled query plans that use this table are automatically invalidated. (In addition, dynamic column addition or removal is an operation that should be audited, at the very least – column removal can cause data loss – and allowing an application to modify a table without maintaining some kind of a trail – including a justification for the action – is not good software practice. Such a feature therefore invites abuse and should be used infrequently and judiciously.)
- Another major limitation is that SQL constraints (e.g., range checks) cannot be applied to sparse columns: the only check that is applied is for correct data type. These would have to be implemented in metadata tables and middle-tier code, as is done in production EAV systems. (This consideration also applies to business applications as well.)
- SQL Server has limitations on row size if attempting to change the storage format of a column: the total contents of all atomic-datatype columns, sparse and non-sparse, in a row that contain data cannot exceed 8016 bytes if that table contains a sparse column for the data to be automatically copied over. Further, sparse columns that happen to contain data have a storage overhead of 4 bytes per column in addition to storage for the data type itself (e.g., 4 bytes for datetime columns). This impacts the amount of sparse-column data that you can associate with a given row. This size restriction is relaxed for the varchar data type, which means that, if one hits row-size limits in a production system, one has to work around it by designating sparse columns as varchar even though they may have a different intrinsic data type. Unfortunately, this approach now subverts server-side data-type checking.

For more details, see http://msdn.microsoft.com/en-us/library/cc280604.aspx

Alternative XML

In the book *Inside Microsoft SQL Server 2008: T-SQL Programming* (Microsoft Press) by Itzik Ben-Gan and other authors, Dejan Sarka (author of the chapter on XML) provides an example of an Open Schema implementation using an XML column in a table to capture the variable/sparse information. This solution, which implements XML-schema-based validation, improves on a simpler version in the 2005 edition of this book, which did not implement validation.

This book provides an overview of approaches to modeling sparse attributes, and points out that building an application that has to manage data gets extremely complicated when using EAV models, because of the extent of infrastructure that has to be developed in terms of metadata tables and application-framework code. The author's XML example (which the authors advocate as simpler than building a framework on top of an EAV design) solves the problem of server-based data validation (which must be done by middle-tier and browser-based code in EAV-based frameworks), but has the following drawbacks:

- The example provided is programmer-intensive: because XML schemas are notoriously tricky to write by hand, the author recommends creating them by defining relational tables, generating XML-schema code, and then dropping these tables. This is problematic in many production operations involving dynamic schemas, where new attributes are required to be defined by power-users who understand a specific knowledge domain (e.g. inventory management or biomedicine) but are not necessarily programmers. In production systems that use EAV, such users define new attributes (and the data-type and validation checks associated with each) through a GUI. (Because the validation-associated metadata is required to be stored in multiple relational tables in a normalized design, a GUI that ties these tables together and enforces the appropriate metadata-consistency checks is the only practical way to allow entry of attribute information, even for advanced developers.)
- The server-based diagnostics that result if incorrect data is attempted to be inserted (e.g., range check or regular-expression pattern violations) are not comprehensible to the end-user: to convey the error accurately, one would, at the least, need to associate a detailed and user-friendly error diagnostic with each attribute.
- The solution does not address the user-interface-generation problem.

All of the above drawbacks are remediable by creating a layer of metadata and application code, but in creating this, the original "advantage" of not having to create a framework has vanished. The fact is that modeling sparse data robustly is a hard database-application-design problem, and there are no shortcuts. Sarka's work, however, proves the viability of using an XML field instead of type-specific relational EAV tables for the data-storage layer, and in situations where the number of attributes per entity is extremely modest (e.g., variable product attributes for different product types) the XML-based solution is more compact than an EAV-table-based one. (XML itself may be regarded as a means of attribute-value data representation, though it is based on structured text rather than on tables.)

EAV and cloud computing

Many cloud computing vendors offer data stores based on the EAV model, where an arbitrary number of attributes can be associated with a given entity. Roger Jennings provides an in-depth comparison[8] of these. In Amazon's offering, SimpleDB, the data type is limited to strings, and data that is intrinsically non-string must be coerced to string (e.g., numbers must be padded with leading zeros) if you wish to perform operations such as sorting. Microsoft's offering, Windows Azure Table Storage, offers a limited set of data types: byte[], bool, DateTime, double, Guid, int, long and string [2] (http://msdn.microsoft.com/en-us/library/dd179338.aspx). The Google App Engine [3] (http://code.google.com/appengine/docs/whatisgoogleappengine.html) offers the greatest variety of data types: in addition to dividing numeric data into int, long, or float, it also defines custom data types such as phone number, E-mail address, geocode and hyperlink. Google, but not Amazon or Microsoft, lets you define metadata that would prevent invalid attributes from being associated with a particular class of entity, by letting you create a metadata model.

Google lets you operate on the data using a subset of SQL; Microsoft offer a URL-based querying syntax that is abstracted via a LINQ provider; Amazon offer a more limited syntax. Of concern, built-in support for combining different entities through joins is currently (April '10) non-existent with all three engines. Such operations have to be performed by application code. This may not be a concern if the application servers are co-located with the data servers at the vendor's data center, but a lot of network traffic would be generated if the two were geographically separated.

An EAV approach is justified only when the attributes that are being modeled are numerous and sparse: if the data being captured does not meet this requirement, the cloud vendors' default EAV approach is often a mismatch for applications that require a true back-end database (as opposed to merely a means of persistent data storage). Retrofitting the vast majority of existing database applications, which use a traditional data-modeling approach, to an EAV-type cloud architecture, would require major surgery. Microsoft discovered, for example, that its database-application-developer base was largely reluctant to invest such effort. More recently, therefore, Microsoft has provided a premium offering – a cloud-accessible full-fledged relational engine, SQL Server Azure, which allows porting of existing database applications with modest changes.

One limitation of SQL Azure is that physical databases are limited to 500GB in size as of January 2015.[9] Microsoft recommends that data sets larger than this be split into multiple physical databases and accessed with parallel queries.

Tree structures and relational databases

There exist several other approaches for the representation of tree-structured data, be it XML or other formats, in a relational database, like Nested set model (de:Nested set, [4] (http://www.kamfonas.com/id3.html)). On the other hand, database vendors have begun to include XML support into their data structures and query features, like in IBM DB2, where XML data is stored as XML separate from the tables, or in PostgreSQL, with a XML data type [5] (http://www.postgresql.org/docs/8.3/interactive/datatype-xml.html) and Xpath queries as part of SQL statements. These developments accomplish, improve or substitute the EAV model approach.

It should be noted, however, that the uses of XML are not necessarily the same as the use of an EAV model, though they can overlap. XML is preferable to EAV for arbitrarily hierarchical data that is relatively modest in volume for a single entity: it is not intended to scale up to the multi-gigabyte level with respect to data-manipulation performance. XML is not concerned per-se with the sparse-attribute problem, and when the data model underlying the information to be represented can be decomposed straightforwardly into a relational structure, XML is better suited as a means of data interchange than as a primary storage mechanism. EAV, as stated earlier, is specifically (and only) applicable to the sparse-attribute scenario. When such a scenario holds, the use of datatype-specific attribute-value tables than can be indexed by entity, by attribute, and by value and manipulated through simple SQL statements is vastly more scalable than the use of an XML tree structure. The Google App Engine, mentioned above, uses strongly-typed-value tables for a good reason.

History of EAV database systems

EAV, as a general-purpose means of knowledge representation, originated with the concept of "association lists" (attribute-value pairs). Commonly used today, these were first introduced in the language LISP.[10] Attribute-value pairs are widely used for diverse applications, such as configuration files (using a simple syntax like *attribute = value*). An example of non-database use of EAV is in UIMA (Uniform Information Management Architecture), a standard now managed by the Apache Foundation and employed in areas such as Natural Language Processing (UIMA). Software that analyses text typically marks up ("annotates") a segment: the example provided in the UIMA tutorial is a program that performs Named Entity Recognition on a document, annotating the text segment "President Bush" with the annotation-attribute-value triple *(Person, Full_Name, "George W. Bush")*.[11] Such annotations may be stored in a database table.

While EAV does not have a direct connection to AV-pairs, Stead and Hammond [12] appear to be the first to have conceived of their use for persistent storage of arbitrarily complex data. The first medical record systems to employ EAV were the Regenstrief electronic medical record (the effort led by Clement MacDonald),[13] William Stead and Ed Hammond's TMR (The Medical Record) system[12] and the HELP Clinical Data Repository (CDR) created by Homer Warner's group at LDS Hospital, Salt Lake City, Utah.[14] (The Regenstrief system actually used a Patient-Attribute-Timestamp-Value design: the use of the timestamp supported retrieval of values for a given patient/attribute in chronological order.) All these systems, developed in the 1970s, were released before commercial systems based on E.F. Codd's relational database model were available, though HELP was much later ported to a relational architecture and commercialized by the 3M corporation. (Note that while Codd's landmark paper was published in 1970, its heavily mathematical tone had the unfortunate effect of diminishing its accessibility among non-computer-science types and consequently delaying the model's acceptance in IT and software-vendor circles. The value of the subsequent contribution of Christopher J. Date, Codd's colleague at IBM, in translating these ideas into accessible language, accompanied by simple examples that illustrated their power, cannot be overestimated.)

A group at the Columbia-Presbyterian Medical Center were the first to use a relational database engine as the foundation of an EAV system.[15]

The open-source TrialDB (http://ycmi.med.yale.edu/TrialDB) clinical study data management system of Nadkarni et al. was the first to use multiple EAV tables, one for each DBMS data type.[2]

The EAV/CR framework, designed primarily by Luis Marenco and Prakash Nadkarni, overlaid the principles of object orientation onto EAV;[3] it built on Tom Slezak's object table approach (described earlier in the "Entity" section). SenseLab (http://senselab.med.yale.edu/), a publicly accessible neuroscience database, is built with the EAV/CR framework. Additionally, there are numerous commercial applications that use aspects of EAV internally including Oracle Designer (applied to ER modeling), Kalido (applied to data warehousing and master data management), and Lazysoft Sentences (applied to custom software development). An EAV system that does not sit on top of a tabular structure but instead directly on a B Tree is InfinityDB, which eliminates the need for one table per value data type.[16]

See also

- Attribute-value system
- Linked Data
- Resource Description Framework (RDF)
- Semantic Web
- Triplestore

References

1. Department of Veterans Affairs: Veterans Health Administration (http://www1.va.gov/health/)
2. Nadkarni, MD, Prakash M.; Marenco, MD, Luis; Chen, MD, Roland; Skoufos, PhD, Emmanouil; Shepherd, MD, DPhil, Gordon; Miller, MD, PhD, Perry (1999), "Organization of Heterogeneous Scientific Data Using the EAV/CR Representation", *Journal of the American Medical Informatics Association* **6** (6): 478–493, doi:10.1136/jamia.1999.0060478 (https://dx.doi.org/10.1136%2Fjamia.1999.0060478), PMC 61391 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC61391), PMID 10579606 (https://www.ncbi.nlm.nih.gov/pubmed/10579606)
3. Marenco, Luis; Tosches, Nicholas; Crasto, Chiquito; Shepherd, Gordon; Miller, Perry L.; Nadkarni, Prakash M. (2003). "Achieving Evolvable Web-Database Bioscience Applications Using the EAV/CR Framework:

Recent Advances", *Journal of the American Medical Informatics Association* **10** (5): 444–53, doi:10.1197/jamia.M1303 (https://dx.doi.org/10.1197%2Fjamia.M1303), PMC 212781 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC212781), PMID 12807806 (https://www.ncbi.nlm.nih.gov/pubmed/12807806)

4. Dinu, Valentin; Nadkarni, Prakash; Brandt, Cynthia (2006), "Pivoting approaches for bulk extraction of Entity–Attribute–Value data", *Computer Methods and Programs in Biomedicine* **82** (1): 38–43, doi:10.1016/j.cmpb.2006.02.001 (https://dx.doi.org/10.1016%2Fj.cmpb.2006.02.001), PMID 16556470 (https://www.ncbi.nlm.nih.gov/pubmed/16556470)
5. GB 2384875 (http://worldwide.espacenet.com/textdoc?DB=EPODOC&IDX=GB2384875), Dingley, Andrew Peter, "Storage and management of semi-structured data", published 6 August 2003, assigned to Hewlett Packard
6. Nadkarni, Prakash (2011), *Metadata-driven Software Systems in Biomedicine*, Springer, ISBN 978-0-85729-509-5
7. Dinu, Valentin; Nadkarni, Prakash (2007), "Guidelines for the effective use of entity-attribute-value modeling for biomedical databases", *International journal of medical informatics* **76** (11–12): 769–79, doi:10.1016/j.ijmedinf.2006.09.023 (https://dx.doi.org/10.1016%2Fj.ijmedinf.2006.09.023), PMC 2110957 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2110957), PMID 17098467 (https://www.ncbi.nlm.nih.gov/pubmed/17098467)
8. Jennings, Roger (2009), "Retire your Data Center", *Visual Studio Magazine*, February 2009: 14–25
9. http://techcrunch.com/2014/04/03/microsofts-azure-sql-can-now-store-up-to-500gb-gets-99-95-sla-and-adds-self-service-recovery/
10. Free Software Foundation (10 June 2007), *GNU Emacs Lisp Reference Manual* (http://www.gnu.org/software/emacs/elisp/html_node/Association-Lists.html), Boston, MA: Free Software Foundation, pp. Section 5.8, "Association Lists"
11. Apache Foundation, UIMA Tutorials and Users Guides. url: http://uima.apache.org/downloads/releaseDocs/2.1.0-incubating/docs/html/tutorials_and_users_guides/tutorials_and_users_guides.html. Accessed Oct 2012,
12. Stead, W.W.; Hammond, W.E.; Straube, M.J. (1982), "A Chartless Record—Is It Adequate?", *Proceedings of the Annual Symposium on Computer Application in Medical Care* **7** (2 November 1982): 89–94, doi:10.1007/BF00995117 (https://dx.doi.org/10.1007%2FBF00995117), PMC 2580254 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2580254)
13. McDonald, C.J.; Blevins, L.; Tierney, W.M.; Martin, D.K. (1988), "The Regenstrief Medical Records", *MD Computing* (5(5)): 34–47
14. Warner, H. R.; Olmsted, C. M.; Rutherford, B. D. (1972), "HELP—a program for medical decision-making", *Comput Biomed Res* **5** (1): 65–74, doi:10.1016/0010-4809(72)90007-9 (https://dx.doi.org/10.1016%2F0010-4809%2872%2990007-9), PMID 4553324 (https://www.ncbi.nlm.nih.gov/pubmed/4553324)
15. Friedman, Carol; Hripcsak, George; Johnson, Stephen B.; Cimino, James J.; Clayton, Paul D. (1990), "A Generalized Relational Schema for an Integrated Clinical Patient Database", *Proceedings of the Annual Symposium on Computer Application in Medical Care* (1990-11-7): 335–339, PMC 2245527 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2245527)
16. http://boilerbay.com/infinitydb/ItemSpaceDataStructures.htm

Further reading

- Pryor, T. (1988), "The HELP medical record system", *M.D. Computing* **5** (5): 22–33, PMID 3231033 (https://www.ncbi.nlm.nih.gov/pubmed/3231033)
- Nadkarni, Prakash, *The EAV/CR Model of Data Representation* (http://ycmi.med.yale.edu/nadkarni/eav_cr_frame.htm), retrieved 1 February 2009
- Dinu, Valentin; Nadkarni, Prakash (Nov–December 2007), "Guidelines for the effective use of entity-attribute-value modeling for biomedical databases", *Int Journal of Medical Informatics* **76** (11–12): 769–779, doi:10.1016/j.ijmedinf.2006.09.023 (https://dx.doi.org/10.1016%2Fj.ijmedinf.2006.09.023), PMC 2110957 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2110957), PMID 17098467 (https://www.ncbi.nlm.nih.gov/pubmed/17098467) Check date values in: `|date=` (help)

- Kyte, Thomas (21 August 2003), *Effective Oracle by Design* (http://asktom.oracle.com/pls/asktom/f?p=100:11:0::::P11_QUESTION_ID:10678084117056), Oracle Press, McGraw-Hill Osborne Media, retrieved 24 August 2007
- Dave M (http://weblogs.sqlteam.com/davidm/articles/12117.aspx)

- The Magento database: concepts and architecture. URL: http://www.magentocommerce.com/wiki/2_-_magento_concepts_and_architecture/magento_database_diagram . Accessed September 2012.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Entity–attribute–value_model&oldid=659629660"

Categories: Database models | Database theory
