IDERA® | Database Tools Whitepaper

THE EAV OF DESTRUCTION

BY JOE CELKO

INTRODUCTION

If you cruise SQL programming newsgroups, you will eventually find a post for what are called EAV ("Entity-Attribute-Value") tables. It is a very common schema design error for programmers who started with an OO language, and now work with NoSQL (attribute, value) pairs or a loosely typed programming language.

The schema looks something like this:

```
CREATE TABLE EAV
(generic_key INTEGER NOT NULL, -- GUID? UUID?
 attribute_name VARCHAR(100) NOT NULL,
 PRIMARY KEY (generic_key, attribute_name),
 attribute_data_type VARCHAR(15) NOT NULL
 CHECK (attribute_data_type IN ('INTEGER', 'FLOAT', etc.)),
 attribute_value VARCHAR(200), -- nullable?
..);
```

This one schema is supposed to model every possible entity in the universe. It is "Automobiles, Squids, and Lady Gaga" all in one. I am showing the generic_key as an INTEGER, but all too often it will be of type BIGINT or GUID. An identifier should not be a numeric in SQL; you do not do math on it, and an identifier is on a nominal scale. If you do not know about the types of scales, you may need a basic data modeling course. The purpose of a GUID or UUID is to locate a Global or Universal resource in cyberspace which is external to the schema. But the EAV programmers use them because they can be generated without having to think about it. GUIDs and BIGINTs from a CREATE SEQUENCE statement or proprietary syntax supposedly do not duplicate. Of course, they eat up disk space, you have no Declarative Referential Integrity (DRI), and you will be casting from VARCHAR(n) to every other data type.

KEY-VALUE FORMAT

The NoSQL people will recognize the (<key>, <value>) model. It has many flavors. This is a quick list of products as of 2019, taken from Wikipedia, using their classifications.

KEY-VALUE CACHE

- Apache Ignite
- Coherence
- eXtreme Scale
- Hazelcast
- Infinispan
- Memcached
- Velocity

KEY-VALUE STORES

- ArangoDB
- Aerospike

EVENTUALLY CONSISTENT KEY-VALUE STORES

- Oracle NoSQL Database
- Amazon DynamoDB
- Riak
- Voldemort

ORDERED KEY-VALUE STORES

- FoundationDB
- InfinityDB
- LMDB
- MemcacheDB

These NoSQL products are designed from the start for handling data in this format. SQL is not.

HOW EAV WORKS

Perhaps it is easier to see with a small example. Let's pretend Archie Andrews (remember the comic books?) needs a database for the girls in his life. Yes, I know that Hugh Hefner might actually need a database, but I want to keep the size of my example small.

```
INSERT INTO
VALUES
(1, 'girlfriend_name', 'Betty Cooper'),
(2, 'girlfriend_name', 'Veronica Lodge'),
(3, 'girlfriend_name', 'Cheryl Blossom');
```

This gets us started by creating an entity identifier. Now we need to add some attributes to the table:

```
INSERT INTO Riverdale
VALUES
(1, 'hair_color', 'Blonde'),
(1, 'income_level', 'Poor'),
(2, 'hair_color', 'Brunette'),
(2, 'income_level', 'Rich'),
(3, 'hair_color', 'Red'),
(2, 'weight', '120'),
(3, 'weight', '130');
```

The generic_key is in one column in Riverdale, so I have to make multiple references to that column to put together the near-equivalent of a three column table.

In most implementations, the only data type in EAV is strings, usually VARCHAR(n) or NVARCHAR(n). It is possible to create a schema where you have numeric, temporal, and string columns for all of the values of those types instead of one super-generic column. Of course, all but one of them will be NULL and all values will be forced into the precision of that data type column. If you so desire, you can enforce data integrity through triggers to ensure that a temporal attribute does not get stored in the String column.

COBOL uses strings for everything, but that language supports regular expressions in the data declarations. This means that money can have a currency symbol, decimal point, and commas provided by the COBOL engine. The closest you can do in SQL is a CHECK () constraint with a SIMILAR TO predicate or a proprietary function to test the data type (ISDATE(), ISNUMERIC(), and so forth which have complications). But now you need to use a CASE expression that is controlled by the attribute_data_type.

But you really want to enforce constraints that are particular to that attribute, such as a list of possible hair colors, numeric ranges, regular expressions, etc. What happens is that the EAV-er does none of this in the database; he shoves it into the applications. Or he does not put it anywhere in the code and the data is a mess.

Try to write a single CHECK () constraint that works for all the parameters. It can be done! You need a CASE expression with WHEN clauses that include every attribute_name as part of the test. For example:

```
CHECK
(CASE
 WHEN attribute_name = ‘weight’
      AND CAST (attribute_value AS INTEGER) > 0
 THEN ‘T’
 WHEN attribute_name = ‘hair_color’
      AND attribute_value IN (‘Blonde’, ‘Brunette’, ‘Red’)
 THEN ‘T’
 WHEN attribute_name = ‘income_level’
      AND attribute_value IN (‘Rich’, ‘Middle’, ‘Poor’)
 THEN ‘T’
 ELSE ‘F’ END = ‘T’)
```

Next, write a CHECK () among two or more parameters with a little math or logic. The WHEN clauses are tested in the order they were written. If you aren’t very careful, you might have them in the wrong order and some of them will never execute.

Try to write a single DEFAULT clause for all the values crammed into one attribute_value column. It is impossible in SQL. DEFAULT is a scalar value and not a CASE expression. But even if a CASE expression were allowed, it would look as bad as the monolithic CASE expression used in the magical, universal, generic CHECK () constraint.

You cannot have a magical, universal DRI action that will CASCADE, SET NULL, RESTRICT and/or SET DEFAULT as needed. That work has to be done with a trigger. If you thought the WHEN clauses in the single CASE expression were unmaintainable, wait until you see the “TRIGGER from Hell” code. Now maintain it. It will not optimize either! Triggers are procedural and not declarative.

You cannot easily guarantee that you get the information you need for each girlfriend. Have you ever asked a girl her weight? Did you notice that we have no value for Betty Cooper’s weight or Cheryl Blossom’s economic status? Should these be NULLs or blanks or a default?

In SQL, you can guarantee a value with a NOT NULL, a DEFAULT and various CHECK () constraints. But here you have to use application level code to force an entry or provide a default value. Let's try doing a simple query.

```
SELECT R0.attribute_value AS girlfriend_name,
    R1.attribute_value AS hair_color
  FROM Riverdale AS R0
    LEFT OUTER JOIN
    Riverdale AS R1
    ON R0.generic_key = R1.generic_key
      AND R0.attribute_name = 'girlfriend_name'
      AND R1.attribute_name = 'hair_color';
```

We have to use outer joins because a value might be missing or (worse) stored with multiple attribute_names so we do not find them. Yes, there is an ISO-11179 rule about data element names having "<attribute>_<attribute property>" syntax, but not everyone follows the standards (i.e. "dob" versus "birth_date") and, if they do, they might not use the same vocabulary ("_dt" versus "_date" for that property).

Since there is no DDL, any typographical error is valid. Old FORTRAN programmers will remember this disaster. The original FORTRAN I and II language had only INTEGER and FLOAT data types. The first occurrence of a variable in the code declared it; if the name began with the letters I thru N, then it was an INTEGER, else it was a FLOAT. Every typo created a new variable!

You have two patterns when you want to add another column, say weight, to the query. You can keep doing outer joins in the same query or you can use a Common Table Expression (CTE). I prefer the CTE for readability, but they both will have the same awful performance.

```
WITH
Col_1
AS
(SELECT R0.attribute_name AS girlfriend_name,
    R1.attribute_name AS hair_color
  FROM Riverdale AS R0
    LEFT OUTER JOIN
    Riverdale AS R1
    ON R0.generic_key = R1.generic_key
      AND R0.attribute_name = 'girlfriend_name'
      AND R1.attribute_name = 'hair_color)

SELECT Col_1.girlfriend_name, Col_1.hair_color,
    CAST (R2.attribute_value AS INTEGER) AS weight
  FROM Col_1
    LEFT OUTER JOIN
    Riverdale AS R2
    ON Col_1.generic_key = R2.generic_key
      AND R2.attribute_name = 'weight');
```

This pattern can be extended for some physical upper limit in the SQL engine. You can use a text editor to “rubber-stamp cut & paste” text for each column into the query.

But if I want to do math on the girl’s weight I also have to cast the string to a numeric of some kind. In this example, I picked INTEGER, but in another query, I could have used FLOAT. In SQL, integer math is not the same as floating point math or decimal math. Picking the correct data types in the DDL is one of the most important design decisions in SQL.

What happens if they give you two weights for one girlfriend? You wind up with multiple rows that make no sense – how can Veronica be both 120 and 150 pounds at once? In SQL, that problem would not exist because a key would allow one and only one value per attribute per row.

When you get a big EAV table and a query with lots of columns, you will not make multiple copies of the table even though that is the conceptual model for aliases. You get pointer structures, working tables, and other devices that you do not see. But they all involve disk access and lots of it. The SQL engine might have to scan the entire table to build each column in each row!

WHY USE EAV?

The most common reasons are:

- You do not have a NoSQL or XML tool in the shop.
- You have no idea what the data is going to be like.
- The data is a fruit salad of things.

EAV is a terrible idea in most cases, and in the first case, you need to actually get a key-value system. As Mark Twain once said, the pumpkin is largely a failure as a shade tree.

In the second case, the EAV is being used as a “breadboard circuit” to test a concept. Frankly, I would rather write a small SQL database and do a careful design from the start. But some people work differently and like a deferred design.

The third case is more likely. If you are a large retail store, your inventory is made up of hundreds or thousands of items. Each item will fall into a general shop category (household goods, clothing, and so forth). Each category will have very different attributes; a refrigerator has electrical specs, while a pair of sneakers has a shoe size, but not vice versa.

If all the goods in the inventory were keyed on their UPC, EAN, or GTIN barcode, then a more normalized design would be a NULL-able column for each attribute and the expectation that most of the columns in a single row are NULL. I have seen this design used with over 1000 columns for an allergy testing database. Each column is the result of a particular test, each with its own particular scale. The key is the patient identifier. There are the expected tests: blood pressure, heart rate, body temperature, etc. But the tests for allergies are done by marking a grid on the patient’s back for a hundred or so tests, sticking the person with a sample of the allergen in grid cell, then waiting to see what happens. The whole catalog of allergens is never used (why check for polar bear fur dander in Hawaii?). The particular patient’s test suite is pulled with a simple CREATE VIEW statement.

WHERE TO GO FROM HERE

The advantage of having everything in one row is that statistical analysis is easier, particularly correlations. However, ~90% of each row will be NULL. This used to be a serious problem, but we have two solutions in modern databases. Not all SQL products will have these features, but you can check for them.

The first is columnar data stores. That is a whole topic by itself, but briefly, each column is kept in a file structure of its own and the rows are re-assembled from them, column by column as needed. Because the column has one and only one data type and most often a limited set of values, it is easy to compress. Those NULLs are such a compressible value.

The second method is to declare the column to be sparse. This feature actually goes back to FORTRAN, when a numeric array could be mostly zeroes. The sparse arrays used special mathematical operators in the compiler to do calculations, and hide all of this from the programmer.

The same idea is used with NULLs in the SQL products that support it. You simply add the key word SPARSE to the column declaration and the SQL will handle the details for you. It is based on a hidden XML implementation. This first appeared in Microsoft SQL Server 2008 and they have a guide for when and how to use it.

You can also read more about working with EAV in Dr. Greg Low's blog, Part 1 and Part 2.

ABOUT THE AUTHOR

Joe Celko is the author of a series of ten books on SQL and RDBMS (MKP/Elsevier) that have been in print for over 20 years. He served for 10 years on the ANSI/ISO database standards committee. He has written columns and articles for the IT trade press for over 30 years. He currently enjoys being a TEALS volunteer and judging the local High School Science Fest once a year.