THE SQL Server Blog Spot on the Web

now hosted by MaximumASP

Aaron Bertrand

Aaron is a senior consultant for SQL Sentry, Inc., makers of performance monitoring and event management software for SQL Server, Analysis Services, and Windows. He has been blogging here at sqlblog.com since 2006, focusing on manageability, performance, and new features; has been a Microsoft MVP since 1997; tweets as @AaronBertrand; and speaks frequently at user group meetings and SQL Saturday events.

What is so bad about EAV, anyway?

I see a lot of people trash-talking the EAV (entity-attribute-value) model. If you want the full story on EAV, I recommend you read the Wikipedia entry. Basically this is the situation where instead of storing the properties of an entity in a single row, you store it as a set of name-value pairs in multiple rows.

The Problem

I understand the general objections to using the EAV model:

- it is difficult to control the attribute names, and to make them consistent;
- it is difficult to enforce data type integrity and referential integrity;
- it is difficult (and slow) to pivot and/or self-join the values to make a single row;
- it is difficult to make certain attributes mandatory; and,
- we've been trained to view this as the "wrong" way to solve the problem.

But what do you do when you have customers that demand real-time, on-demand addition of attributes that they want to store? In one of the systems I manage, our customers wanted to do exactly this. Since we run a SaaS (software as a service) application, we have many customers across several different industries, who in turn want to use our system to store different types of information about *their* customers. A salon chain might want to record facts such as 'hair color,' 'hair type,' and 'haircut frequency'; while an investment company might want to record facts such as 'portfolio name,' 'last portfolio adjustment date,' and 'current portfolio balance.'

Luckily for us, returning that data in a single row wasn't all that important. Our customers wanted to find all the people with red or auburn hair, to sell them some new redhead-only shampoo; or, target all customers with a portfolio balance of $1 million or more, to try to get them to invest in some new pyramid scheme.

The Non-EAV Solution

Without EAV, what would this design look like? Well, we'd have to add a column to the customers table for each of these attributes. So assuming we are opposed to EAV, and that we segment our data by our customers (which we call clients), then by their customers, we would have something like this:

```
CREATE TABLE dbo.Customers
(
    CustomerID                  INT PRIMARY KEY,
```

```
    ClientID                    INT NOT NULL
                                REFERENCES dbo.Clients(ClientID),
    HairColor                   VARCHAR(10),
    HairType                    VARCHAR(10),
    HaircutFrequency            VARCHAR(32),
    PortfolioName               NVARCHAR(32),
    PortfolioBalance            BIGINT,
    LastPortfolioAdjustmentDate SMALLDATETIME
);
```

Then if we wanted to satisfy those two queries, it could be:

```
SELECT CustomerID, HairColor
  FROM dbo.Customers
  WHERE ClientID = 1
  AND HairColor IN ('Red', 'Auburn');

SELECT CustomerID, PortfolioBalance
  FROM dbo.Customers
  WHERE ClientID = 2
  AND PortfolioBalance >= 1000000;
```

Simple enough. But now what if the salon decides they also want to record their customers' head size? The table gets wider and wider as more attributes are required, and you have to build code that will do this (or, *shudder*, add the column manually):

```
ALTER TABLE dbo.Customers ADD HeadSize TINYINT;
```

(And then any code that needs to be used to retrieve head size for the customer needs to be modified to understand the existence of that column.)

So while this reduces the complexity of self-joining to get multiple attributes as you would need to do in the EAV model, it creates its own performance problem by requiring more and more space for any single row, and potentially page splits and fragmentation as that data gets updated. Never mind that HairColor, for example, is probably of very little interest to anyone outside of our salon customer (though maybe we should aim for some of the dating site business). Personally, I don't think dynamic ALTER TABLE ADD new_attribute_name is the way to go here, as you can see that with many customers requiring their own custom attributes, we will soon be up against the one-row-per-page problem. Can you say "maintenance nightmare"?

The EAV solution

My solution: EAV, baby! We talked to our customers who were driving these requirements, and we came up with a good compromise. We didn't need to support every single data type; just three basic ones: strings, numbers, and dates. To simplify things (and based on a LOT of discussion about use cases) we settled on NVARCHAR(1024), DECIMAL(16, 4), and SMALLDATETIME. These covered just about every scenario we were presented with on what

our customers would want to store for each of their customers. Actually, the biggest string they wanted was 255, but I am a forward-thinker - tomorrow they would want 512, and I didn't want to be constantly incrementing this columns, so I stretched it out - if they want to increase them we can simply adjust the parameters and artificial constraints outside of the schema. Similarly, the date column only needed precision to one day, but I left it open to minutes (kind of by choice, since there is nothing between SMALLDATETIME and DATETIME in 2000/2005; in 2008 I would have used DATE most likely), and the decimal requirement was only two decimal places. You might think that such a big decimal is wasteful, but we are using VARDECIMAL storage, which means that only the big decimals will use that space (last year, I ran some tests that show some favorable results).

[And going back to the NVARCHAR column, with SQL Server 2008 R2, we plan to implement Unicode compression so that we can squeeze a lot more out of our space -- we are I/O-bound, not CPU-bound: see post 1, post 2 and post 3 of my tests of this feature earlier this year.]

So the core table looked like this:

```
CREATE TABLE dbo.Attributes
(
    AttributeID INT PRIMARY KEY,
    ClientID    INT NOT NULL
                REFERENCES dbo.Clients(ClientID),
    [Name]      NVARCHAR(32) NOT NULL UNIQUE,
    DataTypeID  TINYINT NOT NULL
                -- 1 = string, 2 = numeric, 3 = date
);
```

Then the actual data table looked like this:

```
CREATE TABLE dbo.CustomerAttributes
(
    CustomerID   INT NOT NULL
                 REFERENCES dbo.Customers(CustomerID),
    AttributeID  INT NOT NULL
                 REFERENCES dbo.Attributes(AttributeID),
    StringValue  NVARCHAR(1024),
    NumericValue DECIMAL(16,4),
    DateValue    SMALLDATETIME,
    ModifiedDate SMALLDATETIME NOT NULL
                 DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (CustomerID, AttributeID)
);
```

The eventual goal of this was to provide better self-service reporting without having to make custom modifications to the system every time a client has a new requirement. This self-service reporting can now be driven by a very simple drop-down that lists all of the IDs/names from the Attributes table, and offers different choices based on the data type. (For example, if it is a string, they can use =, contains, starts with, ends with, etc.; if it is a date, they can use >=, >, BETWEEN, <, <=, etc.) When they add an attribute, nobody has to go change any of this code

because it is already written using this flexibility; the list of attribute choices just gets longer or shorter as attributes are added or removed. (Sorry, I'm not at liberty to share the C# source for the code that builds the self-service reporting queries. It is complicated, but not rocket surgery. Yes, I am mixing metaphors.)

So, for the above queries, we might end up with something like this instead:

```
SELECT c.CustomerID, HairColor = a.StringValue
  FROM dbo.CustomerAttributes AS a
  INNER JOIN dbo.Customers AS c
  ON a.CustomerID = c.CustomerID
  WHERE c.ClientID = 1
  AND a.AttributeID = 1
  AND a.StringValue IN (N'Red', N'Auburn');

SELECT c.CustomerID, PortfolioBalance = a.NumericValue
  FROM dbo.CustomerAttributes AS a
  INNER JOIN dbo.Customers AS c
  ON a.CustomerID = c.CustomerID
  WHERE c.ClientID = 2
  AND a.AttributeID = 2
  AND a.NumericValue >= 1000000;
```

Yes, these queries are a little more verbose, but they are fairly trivial to generate programmatically, when you have a well-defined underlying structure instead of an endless stream of column names on a wide, ever-growing table.

The grass is always greener...

Does this approach have its downsides? Sure:

- **Indexes on the value columns are only marginally useful, and only in some cases.**

  By logging the actual queries that customers run, and how long they take, it is easy to periodically review the heavy hitters, and consider index or statistics changes that would benefit them. So far this hasn't been necessary; maybe we've just been lucky. We have customers with 100s of millions of rows in this table, and while the queries are not sub-second, since they are running behind the scenes and not in real-time, they are certainly well within our SLAs. Anyway for the ones that take longer, these are not queries that would have been any faster in a wide table either (e.g. WHERE PortfolioName LIKE '%small cap%').

- **It is not trivial to return all of the values for a customer's set of attributes in a single row.**

  This is true, but then again, these are easy enough to transpose at the presentation layer (and that is what we do). There is no reason for us to need to present the data for a customer as single-row structure (YMMV).

- **Pulling multiple rows for a customer is more expensive than pulling a single row.**

  This is also true. Though, you could argue that getting to a wide row to access just one skinny attribute is just as wasteful, especially if you are typically after certain and a small number of attributes with much higher

frequency than larger numbers. This really depends on the table, its index structure, the I/O subsystem, access patterns, buffer pool, etc. But more importantly, since we can't define what a "row" looks like for all customers right now, never mind permanently, this trade-off is worth it to us. Again, YMMV.

- **As outlined above, we can't control the consistency or even the presence of certain attributes across customers.**

  Well, we could, with a lot of cumbersome code. But since attribute names are in complete and utter control of each customer, they can live and die by their own control over them. I didn't want to get into the business of determining which attributes customers wanted to make mandatory; if one customer wanted HairColor to be mandatory, and another didn't, then that is an interesting issue to solve declaratively. The most complicated problem has been when a customer has added two similar attributes, e.g. HairColor and Hair_Color, then applied both attributes to a single customer. The resolution is tedious but logically simple:

  1. pick one to keep (let's say we keep AttributeID = 1, and throw away AttributeID = 2);
  2. for any Customers where only AttributeID = 2 exists, update to AttributeID = 1;
  3. for any Customers where both 1 and 2 exist, keep whichever one has a greater ModifiedDate, delete the older one, and if the one remaining is 2, update to 1;
     3a. in the case of a tie, keep 1 and throw away 2;
  4. finally, delete Attribute 2 from the Attributes table.

Conclusion

I just wanted to shine some light on a case where the EAV model might make sense (other than it being written off as a "rookie or object-oriented mistake"). The viability of this approach will depend heavily on how much of the requirements you can gather up front (and whether they are likely to change over time, as in our case), as well as how you want to balance your maintenance time down the road: writing code during every "enhancement" request vs. occasionally troubleshooting performance. At least this is how my experience has panned out thus far. And if I were to design this system from scratch tomorrow, the design would be quite similar, though I might have a better opportunity in that case to see if sparse columns and filtered indexes might make the non-EAV approaches more attractive. In the meantime, I am donning my flame-resistant suit, as I am sure some purists and/or EAV opponents will come out blazing...

Published Thursday, November 19, 2009 7:41 PM by AaronBertrand
Filed under: entity-attribute-value, EAV, database design

Comment Notification

If you would like to receive an email when updates are made to this post, please register here

Subscribe to this post's comments using RSS

Comments

Jack Corbett said:

I was just discussing EAV with a co-worker today. I've used it successfully in the past, and it would be a good fit for a project I am reviewing, not writing.

November 19, 2009 7:34 PM

jamiet said:

Hi Aaron,

Having built almost exactly the same thing myself a couple of years back I can see where you're coming from. Ours was an MDM (Master Data Management) solution where the requirement was to store ALL of the ever-changing entities in the business unit. Users defined what attributes each entity should have (using a custom UI) and under the covers I would dynamically generate:

1) A view per entity that surfaced the information (essentially pivoted the data)

2) A staging table (with matching metadata to the view) into which an ETL process could load data

3) A sproc that took data from the staging table and stored it in the underlying EAV model

We also allowed users to define hierarchies of these entities using an adjacency-list model that keyed back to the EAV.

It worked pretty well whilst still exhibiting all the minuses that you mentioned above. The biggest problem we had though was that people liked the flexibility and started deploying replicas all over the place which simply exacerbated the problem we were trying to solve in the first place (i.e. reference data silos). After I left they even had project admins having their own versions of it so that they could store project-tracking data in it. The 2008 version of the departmental Access database :)

The project was deemed a success but looking back I wonder if we caused more problems than we solved.

Today I would probably look to solve the problem using Master Data Services which is forthcoming in SQL2008 R2. I have deployed its predecessor, Stratature, and achieved good results (and satisfied customers).

Lastly, a question to you, did you consider sparse columns (SQL2008 feature)?

-Jamie

November 19, 2009 7:48 PM

AaronBertrand said:

I didn't consider sparse columns at the time because the project started during the Yukon beta. :-) Even today though, I think sparse columns wouldn't really solve most of the issues I personally have with the non-EAV solutions. They make space consumption and "width" of the table less material, but you still have to deal with metadata updates (and the subsequent ripple effect) whenever a new attribute is needed.

November 19, 2009 9:13 PM

Adam Machanic said:

Hi Aaron,

I've been doing a lot of work with EAV for the past year and have come up with a very similar solution. One thing I would add is a unique constraint on AttributeId/DataTypeId in the Attributes table, and a DataTypeId in the CustomerAttributes table. Then reference BOTH columns in the foreign key, and create a check constraint that enforces the data type and makes sure that the non-appropriate columns are NULL. This well further help avoid data integrity issues.

November 19, 2009 11:01 PM

Mike C said:

Hey Aaron,

Prior to 2008 I had to do implement similar functionality to XML columnsets, but on 2005. We basically implemented an XML data type column to store dynamic user-defined attribute info. Very simple to implement, although we spent a lot of time designing and testing indexing strategies which turned out to be our biggest concern actually.

November 19, 2009 11:30 PM
JohnC said:

What about storing each datatype in it's own table.. I.E. dbo.CustomerAttributesString, dbo.CustomerAttributesNumber, dbo.CustomerAttributesDate. Since you have to figure out what is the correct column to read when they all are in one table... you might as well just figure out what table to

get the data from.. Then you can have foreign keys...and only have a row when that data type is what you need..

November 20, 2009 11:46 AM

Alex Kuznetsov said:

Hey Aaron,

I loved this post. I agree that EAV has its place as a useful solution. Just a few comments:

Indexes on the value columns are only marginally useful, and only in some cases.

I think this is what filtered indexes are for. You can just filter on AttributeID.

It is not trivial to return all of the values for a customers' set of attributes in a single row.

I think this is a perfect case for PIVOT, and it is rather easy. In some cases it is probably faster to PIVOT and transmit less data over the network. Need to do a few benchmarks though.

Makes sense?

November 20, 2009 1:23 PM

AaronBertrand said:

Alex,

It is still 2005, so can't use filtered indexes. I mentioned at the end of the post that I would look at that were I to design it today instead of 4-5 years ago. The problem with the pivot is that this has to be dynamic. This is fairly easy to write when I know what I'm after but the queries are typically on-demand, self-service queries, so I would have to write a query builder around that, and it was definitely out of scope at the time. Anyway we don't have many cases where we need to return the data that way... that just seems to be a common roadblock to EAV out in the wild.

JohnC,

Yes, that is another way to do it, though it doesn't really gain much over my solution (with Adam's suggested enhancement). Your dynamic SQL just changes from @column_name to @table_name, and now you have three tables to manage instead of 1.

November 20, 2009 1:29 PM

Alexander Kuznetsov said:

Of course, PIVOting uses up some CPU. However, if the data is dense (all the cells in the pivoted result

November 20, 2009 2:06 PM

Brad Schulz said:

Hi Aaron...

A woman from Intuit gave a terrific presentation to our (Silicon Valley) SQL Server User Group on this kind of subject a little over a year ago. They experienced the same problems and issues with EAV that you outlined, and they decided to go further and store all the attribute-type data in a single XML datatype column.

In short, their query performance was MUCH better (cuz they indexed the XML and they didn't have to do all those nasty JOINs) and they reduced their diskspace from 77.5 GB down to 20 GB, a 74% reduction.

If you're interested, you can see her PowerPoint presentation at this link (in the document section... the entry entitled "Denise McInerney's XML in MY Database Presentation"):

http://www.baadd.org/SQLServer/SiliconValleySQLServerUserGroup/tabid/68/Default.aspx

--Brad

November 20, 2009 2:09 PM

merrillaldrich said:

As always a great, thorough analysis :-). I like that you prefaced your decision with "here's _why_ EAV would fit the problem." The places I have encountered EAV as unhelpful were in situations where pivoting the data turned out to be the main reporting requirement. If an app just picks a few values using the attribute data, then as you demonstrate it can work nicely. On the other hand if your main output is intended to contain all the attributes for all the entities, then the cost of complicated pivoting selects can outweigh the concerns around adding tables and columns. Here's a case I think might be a counter-example: http://sqlblog.com/blogs/merrill_aldrich/archive/2009/10/29/using-historical-perf-counters-for-storage-planning.aspx

November 20, 2009 2:32 PM

Adam Machanic said:

Brad: With regard to XML property bags, yes--that can be a very good solution. But as with everything else, It Depends -- in this case the question is "what are you doing with the data?" If you just want to send a list of property values back to the app, XML is probably a great choice. If you want to actually query the data in the database, XML may fall quite flat and you may find yourself spending an inordinate amount of time trying to tune monster queries. Pick your poison...

November 20, 2009 2:44 PM

Brad Schulz said:

Now that I look a little closer at your EAV queries, I see this, for example:

SELECT c.CustomerID, HairColor = a.StringValue

FROM dbo.CustomerAttributes AS a

INNER JOIN dbo.Customers AS c

ON a.CustomerID = c.CustomerID

WHERE c.ClientID = 1

AND a.HairColor IN ('Red','Auburn');

But there's no "HairColor" column in CustomerAttributes. Shouldn't the query instead be this?:

SELECT c.CustomerID, HairColor = a.StringValue

FROM dbo.CustomerAttributes AS a

INNER JOIN dbo.Customers AS c

ON a.CustomerID = c.CustomerID

WHERE c.ClientID = 1

AND a.AttributeID = 3 --Whatever Hair Color's ID is

AND a.StringValue IN ('Red','Auburn');

Or, if you don't know what the Hair Color AttributeID is, then you'll have to JOIN in the Attributes table and add a new predicate, like so:

SELECT c.CustomerID, HairColor = a.StringValue

FROM dbo.CustomerAttributes AS a

INNER JOIN dbo.Customers AS c

ON a.CustomerID = c.CustomerID

INNER JOIN dbo.Attributes AS t

ON a.AttributeID = t.AttributeID

WHERE c.ClientID = 1

AND t.Name = 'Hair Color'

AND a.StringValue IN ('Red','Auburn');

...or I guess you could do a scalar subquery (excuse the formatting):

SELECT c.CustomerID, HairColor = a.StringValue

FROM dbo.CustomerAttributes AS a

INNER JOIN dbo.Customers AS c

ON a.CustomerID = c.CustomerID

WHERE c.ClientID = 1

AND a.AttributeID =

(SELECT AttributeID

FROM dbo.Attributes

WHERE Name='Hair Color')

AND a.StringValue IN ('Red','Auburn');

--Brad

November 20, 2009 5:05 PM

AaronBertrand said:

You're totally right Brad, I lifted the code from the first example and didn't finish modifying it. I've corrected it.

In my real use case the join against the Attributes table is not necessary because the query builder is fed the ID. But if you didn't have a query builder, yes, you would need to find some way to translate the user's choice into the AttributeID.

November 20, 2009 5:18 PM

Merrill Aldrich said:

So, found this incredible freeware program on Codeplex with a really simple UI to handle performance

November 20, 2009 6:54 PM

Chris Sherlock said:

Looks like you have an impersonation spammer.

November 22, 2009 6:30 AM

Mladen Prajdic said:

have you thought about using XML datatype with xml schemas to have type safety?

i played with it a while back:

http://weblogs.sqlteam.com/mladenp/archive/2006/10/14/14032.aspx

November 22, 2009 10:04 AM

AaronBertrand said:

Chris Sherlock, what do you mean?

November 22, 2009 12:31 PM

AaronBertrand said:

Mladen, XML might be one way but there seemed to be a lot of overhead to storing all of the tag data also. Besides, when I look over that article, it seems that it relies on a relatively static schema, and that it would be difficult to adapt to a "schema" that changes many times daily...

November 22, 2009 12:32 PM

Peter said:

Chris probably meant "merrillaldrich" != "Merrill Aldrich", with the second touting some product of his.

November 23, 2009 6:52 AM

Aaron Bertrand said:

I think if you follow the link (Merrill's name) you'll see it's just a trackback from one of Merrill's recent blog posts...

November 23, 2009 7:03 AM

Erica Berts said:

I realize this is over a year old, but I would like to know your views on using this EAV approach with cross referencing and hierarchical situations. Cross referencing: Say we have object1 and object2, and they are related somehow. One approach is to have an xref table. Is that what you would advocate in this situation, and why? Hierarchical: Say we have object1 that can have zero or more object2's, and each object2 can have zero or more object3's. What would recommend?

December 20, 2010 11:05 AM

alexeia said:

I have worked with EAV back in 90-s on SQL 7.0/2000 on a real estate project (with original design done by MS Consulting) (for a known real estate agency). At that time we did not know the term EAV and called it "vertical data sets". The DB design had additionally to the described above: subdevision of attributes into functional groups, groups had names, each attribute had name and description, there was configurable functionality of sequence/ordering for groups, and attributes inside groups, each group and each attribute could be mandatory/optional (optional was used on some prod. installations/servers only). There were many hundreds of attributes. The attributes were stored in 6 different native types. There had been also its own data versioning/revisioning system (which included flags: verified/unverified data = approved by a busn. admin on a site). Data was stored and extracted by Userid + AtrributeID PK, CL. Certainly, there was no pivot function at that time, functional pivoting was reached by a MAX from GROUP BY (Userid, ATTRID) inside view. Speed of data extraction (and inserts) was satisfactory (subsecond). Communication with app. was done with vertical recordsets, reporting with horizontal (pivoted) values, specific for each installation (=client). There was an editor for attributes layouts (which allowed to rename and decomission/logically delete them, change sequence inside produced vertical sets etc).

EAV is getting more popular lately due to abundance of very sparse data (medical tests etc).

Alexei Akimov

July 22, 2013 11:28 AM

Leave a Comment

Name (required)

Comments (required)

Remember Me?

About AaronBertrand

...about me...

©2006-2012 SQLblog.com™
Brought to you by Adam Machanic & Peter DeBetta

Contact Us Privacy Statement