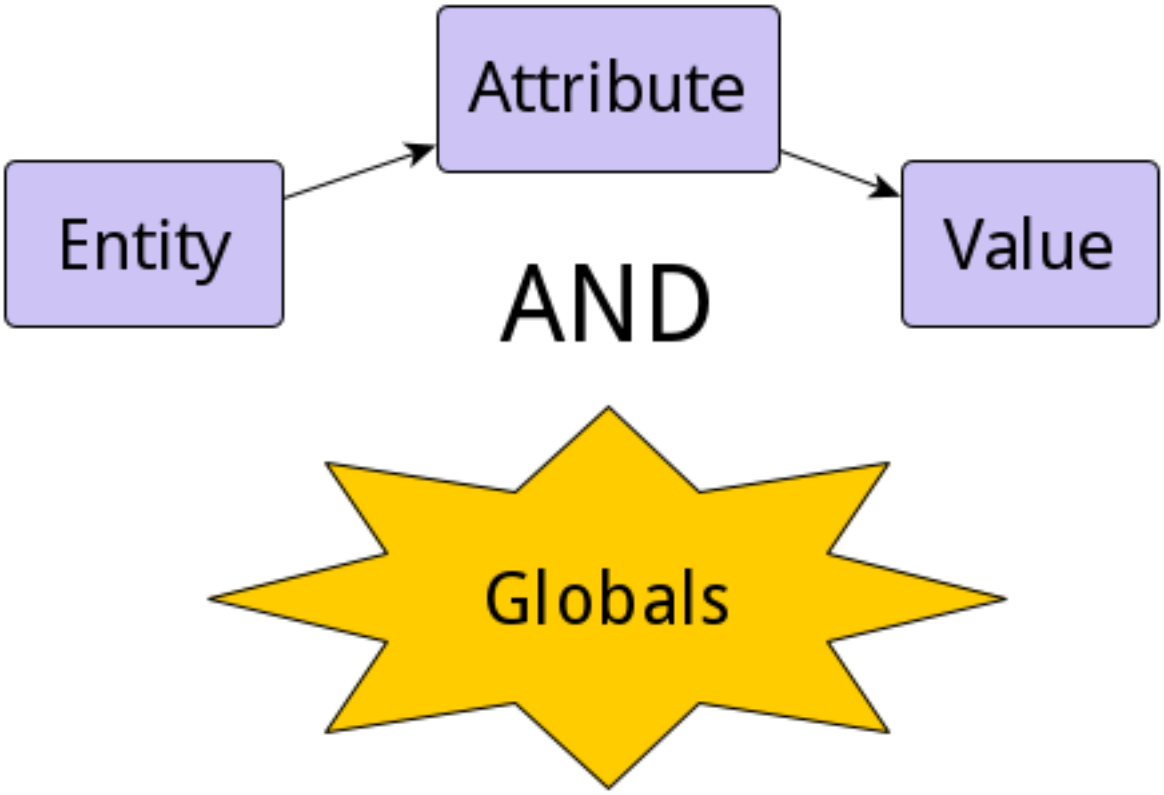
Article
Sergey Kamenev · May 11, 2020 8m read

Open Exchange

Entity-attribute-value model in relational databases. Should globals be emulated on tables? Part 1.



Introduction

In the first article in this series, we ' ll take a look at the entity–attribute–value (EAV) model in relational databases to see how it ' s used and what it ' s good for. Then we'll compare the EAV model concepts to globals.

Sometimes you have objects with an unknown number of fields, or perhaps hierarchically nested fields, for which, as a rule, you need to search.

Take, for example, an online store with diverse groups of products. Each product group has its own unique set of properties and has common properties as well. For example, SSD and HDD drives have the common property " capacity, " while both also have unique properties, " Endurance, TBW " for SSDs and " average head positioning time " for HDDs.

In some situations, the same product, produced by different manufacturers, has its own unique properties.

So, let's imagine we have an online store that sells 50 different groups of goods. Each product group has its own five unique properties, which can be numeric or text.

If we create a table in which each product has 250 properties, though only five are really used, we not only greatly increase (50 times!) the requirements for disk space, we also greatly reduce the speed characteristics of the database, since the cache will be clogged with useless, empty properties.

But that ' s not all. Every time we add a new product group with its own properties, we need to change the structure of the table using the **ALTER TABLE** command. On large tables, this operation might take hours or days, which is unacceptable for business.

" Yes, " the attentive reader will note, " but we can use a different table for each group of products. " Of course, you're right, but this approach gives us a database with tens of thousands of tables for a large store, which is difficult to administer. Moreover, the code, which needs to be supported, becomes increasingly complex.

On the other hand, there ' s no need to change the structure of the database when adding a new group of products. You only need to add a new table for a new group of products.

In any case, users need to be able to easily search the products in a store, get a convenient tabular display of goods showing their current properties, and also be able to compare products.

As you can imagine, a search form with 250 fields would be extremely inconvenient for the user, as would seeing 250 columns of various properties in the product table when only five properties for the group are needed. The same applies to product comparisons.

A marketing database might be another useful example. For each person stored in it, you ' d need many properties (often nested) that might be constantly added, changed, or removed. A person in the past might have bought something for a certain amount, or bought certain groups of goods, participated somewhere, worked somewhere, has relatives, lives in this city, belongs to a certain class of society, and so on and on. There could be thousands of possible fields, constantly changing. Marketers are always thinking about how to distinguish different groups of customers and make compelling special offers to them.

To solve these problems and at the same time have a clear and definite database structure, the entity–attribute–value approach was developed.

The EAV Approach

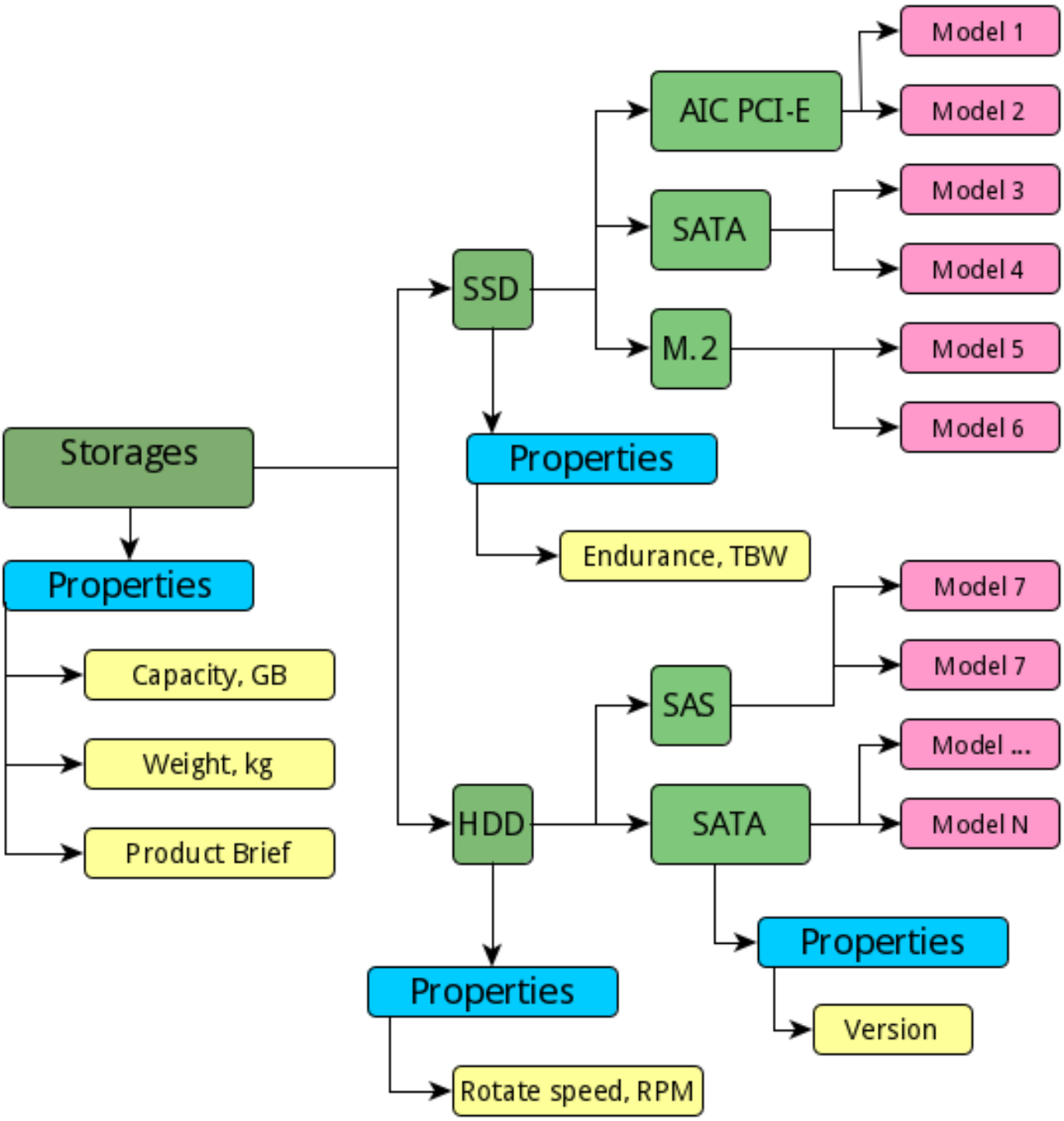
The essence of the EAV approach is the separate storage of entities, attributes, and attribute values. Typically, to illustrate the EAV approach, only three tables are used, called Entity, Attribute, and Value:

Entity	
Id	Name
1	SSD
2	HDD
3	CD

Attribute			
Id	Entity	Name	isNum
1	1	Capacity, GB	1
2	1	Endurance, TWB	1
3	3	Color	0

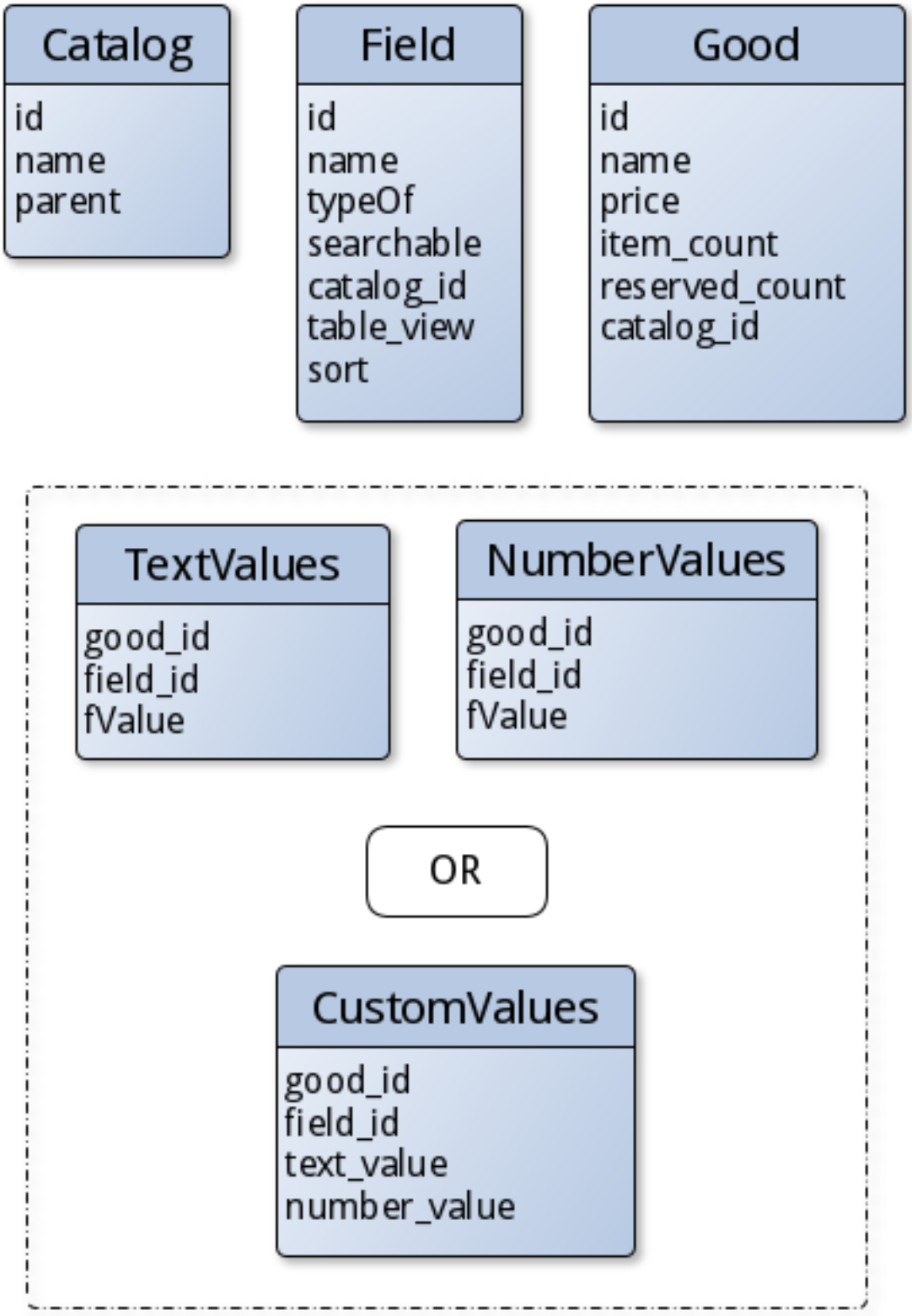
Value				
Instance_id	Entity_id	Attribute_id	Text_value	Num_value
1	1	1		480
2	2	1		2000
1	1	2		1800
3	3	1		800
3	3	3	silver	

The structure of the demo data that we will store.



Implementing the EAV Approach Using Tables

Now let ' s consider a more complex example using five tables (four if you choose to consolidate the last two tables into one).



The first table is Catalog:

```
CREATE TABLE Catalog (
id INT,
name VARCHAR (128),
parent INT
);
```

This table actually corresponds to Entity in the EAV approach. It will store sections of the hierarchical catalog of goods.

The second table is Field:

```
CREATE TABLE Field (
id INT,
name VARCHAR (128),
typeOf INT,
searchable INT,
catalog_id INT,
table_view INT,
sort INT
);
```

In this table, we specify the name of the attribute, its type, and whether the attribute is searchable. We also indicate the section of the catalog that holds the goods to which these properties belong. All products in the catalog section of catalogid or lower might have different properties that are stored in this table.

The third table is Good. It ' s designed to store goods, along with their prices, the total quantity of the goods, the reserved quantity of the goods, and the name of the goods. In principle, you don ' t really need this table but, in my opinion, it ' s useful to have a separate table for the goods.

```
CREATE TABLE Good (
id INT,
name VARCHAR (128),
price FLOAT,
item_count INT,
reserved_count,
catalog_id INT
);
```

The fourth table (TextValues) and the fifth table (NumberValues) are designed to store the values of text and the numeric attributes of goods, and have a similar structure.

```
CREATE TABLE TextValues ??(
good_id INT,
field_id INT,
fValue TEXT
);

CREATE TABLE NumberValues ??(
good_id INT,
field_id INT,
fValue INT
);
```

Instead of the text and number values tables, you could use a single CustomValues table with this structure:

```
CREATE TABLE CustomValues ??(
good_id INT,
field_id INT,
text_value TEXT,
number_value INT
);
```

I prefer to store different types of data separately as it increases speed and saves space.

Accessing the Data Using the EAV Approach

Let ' s start by displaying the catalog structure mapping using SQL:

```
SELECT * FROM Catalog ORDER BY id;
```

In order to form a tree from these values, some separate code is required. In PHP, it would look something like this:

```
$stmt = $ pdo-> query ('SELECT * FROM Catalog ORDER BY id');
$aTree = [];
$idRoot = NULL;

while ($row = $ stmt->fetch())
{
    $aTree [$row ['id']] = ['name' => $ row ['name']];

    if (! $row['parent'])
      $idRoot = $row ['id'];
    else
      $aTree [$row['parent']] ['sub'] [] = $row['id'];
}
```

In the future, we can simply draw the tree if we start from the root node $aTree[$ idRoot].

Now let ' s get the properties of a specific product.

First, we ' ll get a list of properties specific to this product, then attach to it those properties that are in the database. In real life, not all indicated properties are filled and therefore we ' re forced to use LEFT JOIN:

```
SELECT * FROM
(
SELECT g. *, F.name, f.type_of, val.fValue, f.sort FROM Good as g
INNER JOIN Field as f ON f.catalog_id = g.catalog_id
LEFT JOIN TextValues ??as val ON tv.good = g.id AND f.id = val.field_id
WHERE g.id = $ nGood AND f.type_of = 'text'
UNION
SELECT g. *, F.name, f.type_of, val.fValue, f.sort FROM Good as g
INNER JOIN Field as f ON f.catalog_id = g.catalog_id
LEFT JOIN NumberValues ??as val ON val.good = g.id AND f.id = val.field_id
WHERE g.id = $nGood AND f.type_of = 'number'
) t
ORDER BY t.sort;
```

If we use only one table for storing both numerical and text values, the query is greatly simplified:

```
SELECT g. *, F.name, f.type_of, val.text_value, val.number_value, f.sort FROM Good as
 g
INNER JOIN Field as f ON f.catalog = g.catalog
LEFT JOIN CustomValues ??as val ON tv.good = g.id AND f.id = val.field_id
WHERE g.id = $nGood
ORDER BY f.sort;
```

Now we ' ll get the products in the form of a table contained in the $nCatalog catalog section. First, we get a list of properties that should be reflected in the table view for this section of the catalog:

```
SELECT f.id, f.name, f.type_of FROM Catalog as c
INNER JOIN Field as f ON f.catalog_id = c.id
WHERE c.id = $nCatalog AND f.table_view = 1
ORDER BY f.sort;
```

Then we construct the query to create the table. Suppose for a tabular view we need three additional properties (not counting those in the Good table). To simplify the query, we assume that:

```
SELECT g.if, g.name, g.price,
            f1.fValue as f1_val,
            f2.fValue as f2_val,
            f3.fValue as f3_val,
FROM Good
LEFT JOIN TextValue as f1 ON f1.good_id = g.id
LEFT JOIN NumberValue as f2 ON f2.good_id = g.id
LEFT JOIN NumberValue as f3 ON f3.good_id = g.id
WHERE g.catalog_id = $nCatalog;
```

Pros and Cons of the EAV Approach

The obvious plus of the EAV approach is flexibility. With fixed data structures such as tables, we can afford to store a wide variety of property sets for objects. And we can store different data structures without changing the database schema.

We can also use SQL, which is familiar to a great many developers.

The most obvious minus is the mismatch between the logical structure of the data and its physical storage, which causes various difficulties.

Moreover, the programming often involves very complex SQL queries. Debugging can be difficult as you need to create non-standard tools for viewing EAV data. Also, you might have to use LEFT JOIN queries, which slow down the database.

Globals: An Alternative to EAV

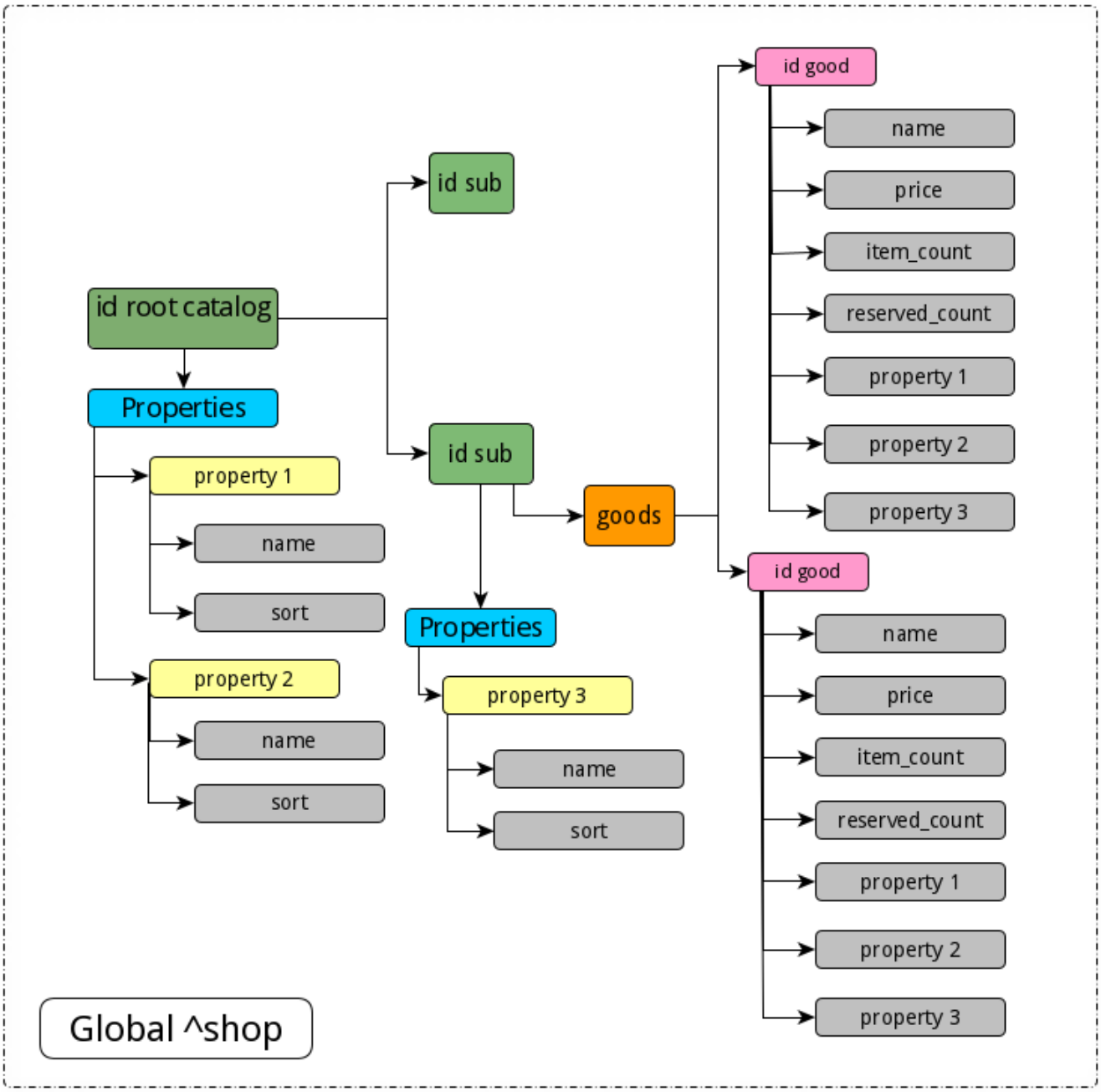
Since I ' m familiar with both the SQL world and the world of globals, I got the idea that using globals for tasks the EAV approach solves would be much more attractive.

Globals are data structures that allow you to store sparse and hierarchical information. A very important point is that globals are carefully optimized for storing hierarchical information. Globals themselves are lower-level structures than tables, which allows them to work much faster than tables.

At the same time, the global structure itself can be selected according to the data structure, which makes the code very simple and clear.

Global Structure for Storing Demo Data

A global is such a flexible and elegant structure for storing data that we could manage with only one global for storing data in catalog sections, properties and products, for example, like this:



Notice how similar the global structure is to the data structure. This compliance greatly simplifies coding and debugging.

In practice, it ' s better to use several globals, although the temptation to store all the information in one is quite strong. It makes sense to make separate globals for indices. You can also separate the storage of the directory partition structure from the goods.

What ' s Next?

In the second article in this series, we ' ll talk about the details and benefits of storing data in InterSystems Iris globals instead of following the EAV model.

#Databases #Globals #Performance #Relational Tables #SQL #Unstructured Data #InterSystems IRIS
Check the related application on InterSystems Open Exchange

Source URL:https://community.intersystems.com/post/entity-attribute-value-model-relational-databases-should-globals-be-emulated-tables-part-1