Datenmodell Entity-Attribute-Value

Holger Jakobs – holger@jakobs.com

2017-03-22

Inhaltsverzeichnis

1 Was ist EAV?

Das Entity-Attribute-Value Model, auch Object-Attribute-Value Model genannt, kommt oft dann zum Einsatz, wenn bezügilch der der Attribute (Eigenschaften) bei Entities (Objekten) gilt:

- Die Anzahl der Attribute variiert sehr stark zwischen den einzelnen Entities (Objekten, Datensätzen).
- Die möglichen Attribute sind zum Zeitpunkt der Schemaerstellung noch nicht bekannt.
- Es können später beliebige weitere Attribute hinzukommen, aber Schemaänderungen sind unerwünscht.

Entity beschreibt hier (wie im Entity-Relationship-Modell bzw. Entity-Relationship-Diagram) die Objekte der Realität, über die Informationen gespeichert werden sollen, beispielsweise Personen, Maschinen, Adressen, Aufträge, Artikel usw.

Gerade bei Artikeln kann man sich vorstellen, dass sie zwar alle eine ganze Reihe von Attributen haben:

- Bezeichnung
- Gewicht
- Verkaufspreis
- aktuelle Lagermenge

Aber darüber hinaus gehende Attribute unterscheiden sich extrem nach Produktgattung. Während Kleidung Eigenschaften über Stoff, Muster, Farbe, Schnitt und Größe hat, haben Smartphones Bildschirmdiagonale, Prozessor, Arbeitsspeicher, Akkukapazität, Softwareversion, Funkstandards, und Fahrräder Anzahl Gänge, Bremsenart, Rahmengröße und Felgengröße. Eine Tabelle mit allen möglichen Spalten, von denen in fast allen Fällen fast alle Spalten den Wert `NULL` haben, erscheint nicht durchführbar, weil ständig neue Attribute hinzukommen – allein wegen der technischen Entwicklung.

Daher behilft man sich oft mit dem sogenannten **EAV**-Modell. Man erstellt statt einer Tabelle für `artikel` drei Tabellen:

- `artikel (artnr, bezeichnung, gewicht, vkpreis, lagermenge)`
- `attrib (attrnr, attr_name)`
- `artikel_attrib (attrnr, artnr, wert)`

Die Tabelle `artikel` enthält ganz normal im relationalen Modell die Eigenschaften, die (fast?) alle Artikel haben. Darüber hinaus gibt es eine Tabelle `attrib`, die mögliche Attribute mit ihrer Nummer und ihrem Namen enthält. Hier trägt man alle neu hinzukommenden Attribute ein, denn in der dritten Tabelle, `artikel_attrib`, ist jeder Kombination aus `artnr` und `attrnr` ein Wert zugeordnet, sofern dieser vorhanden ist. Die Attributkombination aus den `artnr` und `attrnr` ist Primärschlüssel dieser Tabelle, jeweils für sich sind sie Fremdschlüssel bezüglich der beiden erstgenannten Tabellen. Auf diese Weise kann man keine nicht existierenden Eigenschaften oder nicht existierende Artikelnummern eingeben.

Die Spalte `wert` ist von einem Typ, der möglichst viele verschiedene Werte aufnehmen kann. Da man quasi alles als Text darstellen kann, ist hier `text` häufig. Auf diese Weise ist in der Datenbank keine Typprüfung wie in anderen Spalten mehr möglich.

2 Umsetzungsmöglichkeiten in PostgreSQL

2.1 Tabellenlösung

Am einfachsten löst man alles mithilfe relationaler Tabellen.

```
CREATE TABLE artikel (
  artnr        SERIAL        PRIMARY KEY,
  bezeichnung  VARCHAR(100)  NOT NULL,
```

```
  gewicht      INTEGER       NOT NULL CHECK (gewicht > 0),   -- in Gramm
  vkpreis      NUMERIC(10,2) NOT NULL CHECK (vkpreis > 0.00),
  lagermenge   INTEGER       NOT NULL CHECK (gewicht > 0)    -- in Stück
);

CREATE TABLE attrib (
  attrnr       SERIAL        PRIMARY KEY,
  attr_name    VARCHAR(40)   NOT NULL
);

CREATE TABLE artikel_attrib (
  artnr        INTEGER       REFERENCES artikel,
  attrnr       INTEGER       REFERENCES attrib,
  wert         TEXT          NOT NULL,
  PRIMARY KEY (artnr, attrnr)
);
```

Nun können alle Artikel und mögliche Attribute eingetragen werden. Nur wenn diese vorhanden sind, ist es möglich, die zusätzlichen Attribute eines Artikels in die dritte Tabelle einzutragen. Diese Sicherheit wird durch die Fremdschlüsselbeziehungen zwischen den Tabellen abgesichert.

Das Konzept ist flexibel, weil ohne jede Schemaänderung beliebige neue Attribute hinzugefügt werden können.

Nachteilig ist hingegen, dass man alle drei Tabellen mittels `JOIN` verbinden muss, um an die Daten zu kommen. Fragt man nur die Artikeltabelle ab, ist die Information sehr mager.

```
SELECT * FROM artikel;
```

artnr integer	bezeichnung character varying(100)	gewicht integer	vkpreis numeric(10,2)	lagermenge integer
101	Jeans	3400	79.90	12
102	Jeans	3300	79.90	10
103	Jeans	3250	79.90	13
104	Jeans	3400	79.90	12
105	Jeans	3300	79.90	10
106	Jeans	3250	79.90	13
107	Smart X7	250	159.90	3
108	Smart X4	250	159.90	3
109	Smart X7	250	159.90	3
110	Smart X4	250	159.90	3

Also fügt man alle Tabellen in einer Abfrage zusammen, um sämtliche Information zu erhalten. Dabei entsteht dann aus der im Beispiel sehr geringen Datenmenge bereits eine Tabelle mit 38 Zeilen. In der Praxis würde das noch wesentlich größer.

```
SELECT * FROM artikel NATURAL JOIN artikel_attrib NATURAL JOIN attrib
ORDER BY artnr, attrnr;
```

atti inte	artnr integ	bezeichnu character	gewicht integer	vkpreis numeric(10,2)	lage inte	wert text	attr_name character varying(40)
2	105	Jeans	3300	79.90	10	38/36	Kleidergröße
3	105	Jeans	3300	79.90	10	100% Cotton	Stoffmaterial
1	106	Jeans	3250	79.90	13	blau	Farbe
2	106	Jeans	3250	79.90	13	36/36	Kleidergröße
3	106	Jeans	3250	79.90	13	100% Cotton	Stoffmaterial
1	107	Smart X7	250	159.90	3	schwarz	Farbe
5	107	Smart X7	250	159.90	3	2 GB	Arbeitsspeicher
6	107	Smart X7	250	159.90	3	true	µSD-Kartenslot
7	107	Smart X7	250	159.90	3	2	Anzahl SIM-Slots
8	107	Smart X7	250	159.90	3	12	Bildschirmgröße (cm)

Um jetzt alle Information zu einem Artikel auch in einer Zeile zu erhalten, müsste man die Ergebnisse entsprechend zusammenfassen. PostgreSQL bietet hierfür Lösungen an.

2.1.1 Zusammenfassung in Array

```
SELECT a.*, array_agg(attr_name || ':' || wert) AS attribs
FROM artikel a NATURAL JOIN artikel_attrib NATURAL JOIN attrib
GROUP BY a.artnr
ORDER BY artnr;
```

Zu jedem Artikel gibt es jetzt eine Spalte `attribs`, die so aussieht:

attribs text[]
{Farbe:blau,"Stoffmaterial:100% Cotton",Kleidergröße:33/36}
{Farbe:blau,Kleidergröße:34/34,"Stoffmaterial:100% Cotton"}
{Farbe:blau,"Stoffmaterial:100% Cotton",Kleidergröße:32/36}
{"Stoffmaterial:100% Cotton",Farbe:schwarz,Kleidergröße:32/32}
{"Stoffmaterial:100% Cotton",Farbe:blau,Kleidergröße:38/36}
{Kleidergröße:36/36,Farbe:blau,"Stoffmaterial:100% Cotton"}
{"Bildschirmgröße (cm):12",Farbe:schwarz,"Arbeitsspeicher:2 GB",µSD-Ka
{"Anzahl SIM-Slots:2",Farbe:weiß,µSD-Kartenslot:false,"Bildschirmgrö
{"Bildschirmgröße (cm):15.5",µSD-Kartenslot:true,"Anzahl SIM-Slots:2",
{"Arbeitsspeicher:2 GB",µSD-Kartenslot:true,"Anzahl SIM-Slots:1",Farb

Hierbei werden zwei Besonderheiten von PostgreSQL eingesetzt. Die `GROUP BY`-Klausel enthält nur den Primärschlüssel, obwohl weitere Spalten unaggregiert ausgegeben werden. Falls nach dem PK gruppiert wird, weiß PostgreSQL, dass alle anderen Spaltenwerte ein-

deutig sind und meldet keinen Fehler. Die Funktion `array_agg()` aggregiert alle Werte zu einem Array. Dass von allen Datentypen auch Arrays existieren können, ist nicht Standard.

2.1.2 Zusammenfassung in JSON-Objekt

```
SELECT a.*, json_object_agg(attr_name, wert) AS attribs
FROM artikel a NATURAL JOIN artikel_attrib NATURAL JOIN attrib
GROUP BY a.artnr
ORDER BY artnr;
```

Zu jedem Artikel gibt es jetzt eine Spalte `attribs`, die so aussieht:

attribs json
{ "Farbe" : "blau", "Stoffmaterial" : "100% Cotton", "Kleidergröße" : "33/36" }
{ "Farbe" : "blau", "Kleidergröße" : "34/34", "Stoffmaterial" : "100% Cotton" }
{ "Farbe" : "blau", "Stoffmaterial" : "100% Cotton", "Kleidergröße" : "32/36" }
{ "Stoffmaterial" : "100% Cotton", "Farbe" : "schwarz", "Kleidergröße" : "32/32"
{ "Stoffmaterial" : "100% Cotton", "Farbe" : "blau", "Kleidergröße" : "38/36" }
{ "Kleidergröße" : "36/36", "Farbe" : "blau", "Stoffmaterial" : "100% Cotton" }
{ "Bildschirmgröße (cm)" : "12", "Farbe" : "schwarz", "Arbeitsspeicher" : "2 GB",
{ "Anzahl SIM-Slots" : "2", "Farbe" : "weiß", "µSD-Kartenslot" : "false", "Bildsch
{ "Bildschirmgröße (cm)" : "15.5", "µSD-Kartenslot" : "true", "Anzahl SIM-Slots"
{ "Arbeitsspeicher" : "2 GB", "µSD-Kartenslot" : "true", "Anzahl SIM-Slots" : "1"

Hierbei werden zwei Besonderheiten von PostgreSQL eingesetzt. Die erste wurde im vorigen Abschnitt schon beschrieben. Die Funktion `jsbon_object_agg()` aggregiert alle Key-Value-Paare zu einem JSON-Objekt.

Wesentlicher Vorteil ist, dass hier ein standardisiertes Datenformat (RFC 7159) verwendet wird, das quasi alle Programmiersprachen – insbesondere natürlich JavaScript – leicht verarbeiten können, um die einzelnen Werte herauszulösen.

2.2 HSTORE-Lösung

PostgreSQL bietet an, in einer Spalte eine Liste aus Key-Value-Paaren zu speichern. Dies gibt es schon sehr lange und nennt sich `HSTORE`. Die Werte werden dort in einem speziellen Format eingetragen, bei dem Key und Value jeweils in doppelte Anführungsstrichen geschrieben und mit der Zeichenkombination `=>` getrennt werden. Die einzelnen Paare wiederum werden mit Komma getrennt. Um `HSTORE` benutzen zu können, muss die gleichnamige Extension in der jeweiligen Datenbank erzeugt werden.

So kann eine passende Tabelle erzeugt werden, die dann alle drei bisherigen Tabellen ersetzt.

```
CREATE EXTENSION HSTORE;
```

```
CREATE TABLE artikel_hstore (
  artnr        SERIAL        PRIMARY KEY,
  bezeichnung  VARCHAR(100)  NOT NULL,
  gewicht      INTEGER       NOT NULL CHECK (gewicht > 0),   -- in Gramm
  vkpreis      NUMERIC(10,2) NOT NULL CHECK (vkpreis > 0.00),
  lagermenge   INTEGER       NOT NULL CHECK (gewicht > 0),    -- in Stück
  attribs HSTORE
);
```

Um die bisherigen Werte dort einzutragen, verwendet man folgendes Kommando:

```
INSERT INTO artikel_hstore
SELECT a.*, string_agg('"' || attr_name || '"=>"' || wert || '"', ', ')::HSTORE
FROM artikel a NATURAL JOIN artikel_attrib NATURAL JOIN attrib
GROUP BY a.artnr;
```

Die Daten können ganz schlicht abgefragt werden, wobei dann das etwas spezielle `HSTORE`-Format herauskommt. Möchte man ein eher standardgemäßes Datenformat, dann bietet sich die Konvertierung in JSON an. Allerdings muss man dann alle Spalten, die man gerne hätte, einzeln nennen:

```
SELECT artnr, bezeichnung, to_json(attribs) FROM artikel_hstore;
```

Nachteil von `HSTORE` ist, dass es lediglich Key-Value-Paare gibt, also keine Schachtelung von Strukturen. Genau das bietet – wenn auch im vorliegenden Beispiel nicht genutzt – das JSON-Format. Man könnte sich aber schon vorstellen, dass ein Attribut `Maße` sich unterteilt in `Länge`, `Breite` und `Höhe` oder dass die ganzen Eigenschaften von Artikeln ggf. gruppiert werden, so dass man sie in Gruppen abfragen kann.

Bei `HSTORE` kann man auch auf Teile zugreifen, d. h. sie sind auf jeden Fall günstiger als eine reine Textspalte, in der man die Daten in einem dem Datenbanksystem unbekannten Format ablegt. So kann man schnell alle Artikel heraussuchen, die das Attribut „Kleidergröße“ haben – also offensichtlich Kleidungsstücke sind.

```
SELECT artnr, bezeichnung,
  attribs->'Kleidergröße' AS "Größe",
  attribs->'Stoffmaterial' AS "Material"
FROM artikel_hstore
WHERE attribs ? 'Kleidergröße';
```

	artnr integer	bezeichnung character varying(100)	Größe text	Material text
1	106	Jeans	36/36	100% Cotton
2	103	Jeans	32/36	100% Cotton
3	104	Jeans	32/32	100% Cotton
4	102	Jeans	34/34	100% Cotton
5	105	Jeans	38/36	100% Cotton
6	101	Jeans	33/36	100% Cotton

Auch ist es leicht möglich, alle blauen Kleidungsstücke zu finden, oder auch alle mit Länge 36:

```
SELECT artnr, bezeichnung,
  attribs->'Kleidergröße' AS "Größe", attribs->'Farbe' AS "Farbe"
FROM artikel_hstore
WHERE attribs ? 'Kleidergröße' AND attribs @> '"Farbe"=>"blau"';

SELECT artnr, bezeichnung, attribs->'Kleidergröße' AS "Größe"
FROM artikel_hstore
WHERE attribs->'Kleidergröße' ~ '/36$';
```

An Besonderheiten von PostgreSQL kommt hier neben den `HSTORE`-Funktionen noch der „matches“-Operator zum Einsatz, der auf reguläre Ausdrücke prüft und als Tilde (~) geschrieben wird.

2.3 JSON-Lösung

Was spricht dagegen, die Speicherung so, wie sie in Abschnitt 2.1.2 auf Seite 5 ausgegeben wurde, gleich als Tabelle festzulegen?

```
CREATE TABLE artikel_json (
  artnr        SERIAL        PRIMARY KEY,
  bezeichnung  VARCHAR(100)  NOT NULL,
  gewicht      INTEGER       NOT NULL CHECK (gewicht > 0),   -- in Gramm
  vkpreis      NUMERIC(10,2) NOT NULL CHECK (vkpreis > 0.00),
  lagermenge   INTEGER       NOT NULL CHECK (gewicht > 0),    -- in Stück
  attribs JSON
);
```

Tatsächlich kann man die Ergebnisse der im Abschnitt 2.1.2 gezeigten Abfrage unmittelbar in diese neue Tabelle übertragen (lediglich die Sortierung wurde weggelassen, weil unnötig).

```
INSERT INTO artikel_json
SELECT a.*, json_object_agg(attr_name, wert)
FROM artikel a NATURAL JOIN artikel_attrib NATURAL JOIN attrib
GROUP BY a.artnr;
```

Also bietet es sich geradezu an, auf die beiden weiteren Tabellen zu verzichten. Allerdings kann man dann beliebige Attributnamen eintragen, weil es keinen Fremdschlüssel mehr auf eine Tabelle mit den möglichen Attributen gibt. Das ist aber in vielen Fällen durchaus in Ordnung, weil man hier möglichst große Freiheiten will.

Neben JSON bietet PostgreSQL auch die Alternative JSONB. Beim Einlesen von Daten ist sie etwas langsamer, aber dafür ist es möglich, über Teile der JSON-Struktur Indexe zu bilden, Vergleiche durchzufüren und auch einfacher, Teile herauszulösen. Von der Vorgehensweise bis hierher ist es völlig identisch mit dem oben gezeigten, es ist lediglich der Datentyp von `JSON` in `JSONB` zu ändern.

Nur bei `JSONB` kann man so schnell alle Artikel heraussuchen, die das Attribut „Kleidergröße“ haben – also offensichtlich Kleidungsstücke sind.

```
SELECT bezeichnung,
  attribs->>'Kleidergröße' AS "Größe",
  attribs->>'Stoffmaterial' AS "Material"
FROM artikel_jsonb
WHERE attribs ? 'Kleidergröße';
```

	bezeichnung character varying(100)	Größe text	Material text
1	Jeans	33/36	100% Cotton
2	Jeans	34/34	100% Cotton
3	Jeans	32/36	100% Cotton
4	Jeans	32/32	100% Cotton
5	Jeans	38/36	100% Cotton
6	Jeans	36/36	100% Cotton

Auch ist es wie beim `HSTORE` leicht möglich, alle blauen Kleidungsstücke zu finden, oder auch alle mit Länge 36:

```
SELECT artnr, bezeichnung,
  attribs->>'Kleidergröße' AS "Größe", attribs->>'Farbe' AS "Farbe"
FROM artikel_jsonb            -- geht nicht mit JSON
WHERE attribs ? 'Kleidergröße' AND attribs @> '{"Farbe":"blau"}';

SELECT artnr, bezeichnung,
  attribs->>'Kleidergröße' AS "Größe", attribs->>'Farbe' AS "Farbe"
FROM artikel_json            -- geht auch mit JSONB
WHERE attribs->>'Kleidergröße' IS NOT NULL AND attribs->>'Farbe' = 'blau';

SELECT artnr, bezeichnung, attribs->>'Kleidergröße' AS "Größe"
FROM artikel_jsonb            -- geht auch mit JSONB
WHERE attribs->>'Kleidergröße' ~ '/36$';
```

3 Fazit

Es gibt mehrere gangbare Wege, aber unter Ausnutzung der Eigenschaften von PostgreSQL erscheint die `JSONB`-Lösung die flexibelste und gleichzeitig auch performanteste. Allerdings wurde dieser Datentyp erst mit PostgreSQL Version 9.4 eingeführt. Wer alte Anwendungen hat, die `HSTORE` verwenden, ist jetzt nicht gezwungen, diese umzustellen. Neuentwicklungen sollten aber wegen der größeren Flexibilität auf `JSONB` setzen. Dass man `HSTORE` als Erweiterung laden muss und es nicht im Kern enthalten ist, stellt kein Problem dar.