Case study **6**

Schema-less databases - Part 3

Objective: This document studies a fourth family of alternative data models, namely the *object* (or *document)* models. In these models, the information is represented by complex objects in which properties can be multivalued and composite.

Keywords: non-relational data model, key-value model, object model, NoSQL, schema-less database, multivalued property, composite property, document-oriented DBMS, MongoDB, CouchDB, Azure, Datastore Oracle, metadata, index, data migration, schema conversion

6.1 Object view of Key-Value data

At the opposite of the *Key-Value* view of the data, the *object view* aggregates all the data of each entity as a complex object associated with this entity. If we imagine these data stored in a table, then each entity is described by a single (but complex) row.

An interesting property of the *Key-Value* model developed in Part 2 (that comprises table M_CUSTOMER for example) is that one can easily use it derive this object[1] view. With the object model, application programs read **complex records** each comprising an entity Id (represented by, say, column **Entity**) and a list of *Key-Value* couples represented by column **Attributes** (plural!).

Figure 6.1 shows some excerpts from a table in which the data of source table CUSTOMER are structured as objects. This table has been generated by Script 6.1 from *Key-Value* table M_CUSTOMER.

Entity	Attributes
B062	{"Account":"-3200.0","Address":"72, r. de la Gare","Cat":"B2", "City":"Namur","CustID":"B062","Name":"GOFFIN"}
B112	{"Account":"1250.0","Address":"23, r. Dumont","Cat":"C1", "City":"Poitiers","CustID":"B112","Name":"HANSENNE"}
...	...
C123	{"Account":"-2300.0","Address":"25, r. Lemaître","Cat":"C1", "City":"Namur","CustID":"C123","Name":"MERCIER"}
C400	{"Account":"350.0","Address":"65, r. du Tertre","Cat":"B2", "City":"Poitiers","CustID":"C400","Name":"FERARD"}
D063	{"Account":"-2250.0","Address":"201, bvd du Nord", "City":"Toulouse","CustID":"D063","Name":"MERCIER"}
...	...
K111	{"Account":"720.0","Address":"180, r. Florimont","Cat":"B1", "City":"Lille","CustID":"K111","Name":"VANBIST"}
K729	{"Account":"0.0","Address":"40, r. Bransart", "City":"Toulouse","CustID":"K729","Name":"NEUMAN"}
L422	{"Account":"0.0","Address":"60, r. de Wépion","Cat":"C1", "City":"Namur","CustID":"L422","Name":"FRANCK"}
...	...

Figure 6.1 - Object view of Key-Value data (JSON format)

```
select Entity,
       '{'
       ||group_concat('"'||Attribute||'":"'||Value||'"',',')
       ||'}' as Attributes
from   M_CUSTOMER
group by Entity;
```

Script 6.1 - Generating an object view of Key-Value data (JSON format)

The list of name:value pairs can be built according to a proprietary or standard format such as Key-Value, CSV, XML or JSON (*JavaScript Object Notation*), the format chosen in this example.

We notice that attribute **Cat** does not appear when its value is *null* (in entities D063 and K729). This is consistent with a model according to which all (and only) the information known about the entity is available in its record.

We also observe that the value of attribute **Account** is double-quoted, which is useless since it is a number. This can be fixed easily with Script 6.2.

1. The term *object* refers to the concepts of *Object-oriented* data models, namely the ability to build and manage complex data structures, as compared with the flat structure of standard relational tables. These data aggregates perfectly fit to (true) objects in Java programs for instance, thus reducing the so-called *impedance mismatch* between databases and OO programs without resorting to *Object/Relational Mapping* (O/RM) middleware.

```
select Entity,
       '{'||group_concat('"'||Attribute||'":'
                         ||case when Attribute in ('Account')
                                then Value
                                else '"'||Value||'"'
                           end, ',')
     ||'}' as Attributes
from M_CUSTOMER
group by Entity
```

Script 6.2 - Generating an object view of Key-Value data (JSON format); fixing the quoted representation of *numeric* values.

6.2 Objects with complex properties

So far, the attributes of the objects we have built have elementary values. Now we will examine how to represent and compute complex properties, the instances of which are multivalued, composite or both.

6.2.1 Multivalued properties

We want to add to customer data the list of *amounts* of the orders they have placed. Of course, these amounts can be computed from the data available in the source tables, but here, we want to store them as attributes in table M_CUSTOMER, just to illustrate the point. Script 6.3 will insert these new rows. Of course, customers who have not placed any order also have no Amount attribute.

```
insert into M_CUSTOMER(Entity,Attribute,Value)
select CustID,'Amount',sum(Qord * Price)
from   CUSTORDER O, DETAIL as D, PRODUCT as P
where  O.OrdID = D.OrdID and D.ProdID = P.ProdID
group by CustID,O.OrdID;
```

Script 6.3 - Adding multivalued attribute *Amount* to table M_CUSTOMER

Not surprisingly, this query fails ("*SQL execution error (columns Entity, Attribute are not unique)*"). Indeed, the primary key declared for M_CUSTOMER prevents us from storing more than one attribute with a given name, which make customer C400 uncomfortable, with her three orders! To allow multiple attributes with the same name, we must drop this constraint[2] (Script 6.4). Now, the script works fine (Figure 6.2).

```
create table M_CUSTOMER(
   Entity    varchar(32)   not null,
   Attribute varchar(32)   not null,
   Value     varchar(32)
   primary key (Entity,Attribute));
```

Script 6.4 - Modifying the structure of table M_CUSTOMER to let it accommodate multivalued attributes

Entity	**Attribute**	**Value**
...	...	...
C400	CustID	C400
C400	Name	FERARD
C400	Address	65, r. du Tertre
C400	City	Poitiers
C400	Cat	B2
C400	Account	350
C400	**Amount**	**6400**
C400	**Amount**	**28500**
C400	**Amount**	**315**
F011	CustID	F011
F011	Name	PONCELET
F011	Address	17, Clôs des Erables
F011	City	Toulouse
F011	Cat	B2
F011	Account	0
F011	Amount	169625
...	...	...

Figure 6.2 - A customer can have more than one attribute of the same type

It would be nice if we could produce JSON objects in which, for each customer, the amount values are presented as an **array**, as in Figure 6.3.

```
{"Account":"350", "Address":"65, r. du Tertre", "Cat":"B2",
 "City":"Poitiers", "CustID":"C400", "Name":"FERARD",
 "Amounts":[6400,28500,315]}
```

Figure 6.3 - A complex JSON object with multivalued attribute **Amount**

This is quite possible in SQL, but it is a bit more complex than the generation of flat objects as those in Figure 6.1, because we have to combine two levels of aggregation, the highest for entity grouping and the innermost for array grouping.

2. We could have extended the primary key by adding to it column `Value`. However this would have been valid only for distinct values of this attribute, which is clearly not the case in general for attribute Amount.

The simplest way to proceed is to preprocess the data in such a way that entities have single-valued attributes only, so that standard Script 6.2 applies. This is performed by a **view** (Script 6.5) that first defines new attribute **Amounts** through a `group_concat` function, then adds all the other attributes. Figure 6.4 shows the contents of this view.

Entity	**Attribute**	**Value**
...	...	...
C400	CustID	C400
C400	Name	FERARD
C400	Address	65, r. du Tertre
C400	City	Poitiers
C400	Cat	B2
C400	Account	350
C400	**Amounts**	**[6400,28500,315]**
...	...	...

Figure 6.4 - Collecting the multiple values of **Amount** in an array

```
create temp view CUST_AMOUNT(Entity,Attribute,Value) as
select Entity,'Amounts','['||group_concat(Value,',')||']'
from   M_CUSTOMER
where  Attribute = 'Amount'
group by Entity
  union
select Entity,Attribute,Value
from   M_CUSTOMER
where  Attribute <> 'Amount';
```

Script 6.5 - Transforming sequences of **Amount** values in array **Amounts**

Script 6.6 then extracts the 2-level objects. It is an extension of Script 6.2 in which numeric values are left unquoted.

6.2.2 Composite multivalued properties

Let us now try to generate even more complex objects, such as that of Figure 6.5. In object C400, besides elementary attributes **CustID**, **Name** and **City**, we have introduced attribute **Orders**, which itself is an *array of objects*, each one containing the data of an order of the current customer. This attribute is both *multivalued* and *composite*. It results from the integration of **CUSTORDER** data in the object representing a customer.

```
select Entity,
       '{'
       ||group_concat('"'
                ||Attribute
                ||'":'
                ||case when Attribute in ('Account','Amounts')
                       then Value
                       else '"'||Value||'"'
                  end,
                ',')
       ||'}' as Attributes
from  CUST_AMOUNT
group by Entity;
```

Script 6.6 - Producing complex JSON objects including an array

```
{"CustID":"C400","Name":"FERARD","City":"Poitiers",
 "Orders":[{"OrdID":"30179","DateOrd":"2015-12-22"},
           {"OrdID":"30184","DateOrd":"2015-12-23"},
           {"OrdID":"30186","DateOrd":"2016-01-02"}]
}
```

Figure 6.5 - A JSON object comprising an array of composite objects

We could extract the information from Key-value table M_CUSTORDER but is easier to start from source table CUSTORDER. Script 6.7 defines view **CUST_ORDERS** that returns the data of Figure 6.6.

```
+--------+-----------+-----------------------------------------+
| Entity | Attribute | Value                                   |
+--------+-----------+-----------------------------------------+
| B512   | Orders    | [{"OrdID":"30188","DateOrd":"2016-01-03"}] |
| C400   | Orders    | [{"OrdID":"30179","DateOrd":"2015-12-22"}, |
|        |           |  {"OrdID":"30184","DateOrd":"2015-12-23"}, |
|        |           |  {"OrdID":"30186","DateOrd":"2016-01-02"}] |
| F011   | Orders    | [{"OrdID":"30185","DateOrd":"2016-01-02"}] |
| K111   | Orders    | [{"OrdID":"30178","DateOrd":"2015-12-21"}] |
| S127   | Orders    | [{"OrdID":"30182","DateOrd":"2015-12-23"}] |
+--------+-----------+-----------------------------------------+
```

Figure 6.6 - Build the values of attribute Orders as an array of JSON objects

```
create temp view CUST_ORDERS(Entity,Attribute,Value) as
select CustID,'Orders',
     '['||group_concat('{'
        ||'"OrdID":"'||OrdID||'","DateOrd":"'||DateOrd||'"'
        ||'}', ',')||']'
from   CUSTORDER
group by CustID;
```

Script 6.7 - Creating attribute **Orders** for each customer entity (customers who have not placed orders are not represented)

To build the objects of Figure 6.5, we must slightly adapt the generation query so that it works on the *union* of **M_CUSTOMER** and **CUST_ORDERS** (Script 6.8).

```
select Entity,
       '{'
       ||group_concat('"'
                      ||Attribute
                      ||'":'
                      ||case when Attribute in ('Orders','Account')
                             then Value
                             else '"'||Value||'"'
                        end,
                      ',')
       ||'}' as Attributes
from  (select Entity,Attribute,Value
       from M_CUSTOMER
         union
       select Entity,Attribute,Value
       from CUST_ORDERS)
where Entity in ('C400','B062','B512')
group by Entity;
```

Script 6.8 - Producing complex JSON objects including an array of objects

6.3 Indexing objects

Let us consider that the JSON objects we have created above are stored in table **CUSTOMER_JSON** with columns **Entity** and **Attributes**. We also suppose that there are millions of rows in this table. Remembering how indexing a relational table is easy through **`create index`** statements, we could worry about how we could index complex, multivalued and/or composite, attributes such as those stored in column **Attributes**. For instance, if we want to retrieve data on the customers who *live in*

Poitiers or who have *placed an order on 2016-01-02*, we have so far no straightforward way to get them but by selecting the objects through, say, a **`like`** predicate. The corresponding query is simple but particularly inefficient, since it requires a full scan of the table:

```
select *
from   CUSTOMER_JSON
where  Attributes like '%"City":"Poitiers"%';
```

Actually, indexing such structures is much easier than we could think. Let us consider table **M_CUSTOMER** again. As is usual in most DBMS, an index is automatically created on its primary key. We decide to create a second index on columns (**Attribute**, **Value**) to get a fast access to all the entities from their attribute values:

```
create index XATTVAL on M_CUSTOMER(Attribute,Value);
```

As a consequence, the following query will return the set of desired **Entity** values corresponding to customers living in Poitiers in a matter of tens of milliseconds:

```
select Entity
from   M_CUSTOMER
where  Attribute = 'City'
and    Value = 'Poitiers';
```

Better, if we are only interested in indexing data on attribute City, we can create a *partial (*or *filtered) index*, which is much smaller and therefore more efficient

```
create index XCITYVAL on M_CUSTOMER(Value)
where Attribute = 'City';
```

We then write the query of Script 6.9, that returns the attribute values of these entities in JSON format.

```
select Entity,Attributes
from   CUSTOMER_JSON
where  Entity in (select Entity
                  from   M_CUSTOMER
                  where  Attribute = 'City'
                  and    Value = 'Poitiers');
```

Script 6.9 - Which customers live in Poitiers?

6.4 Comparison of the data models

We have described a series of alternative data models that offer more generic data structures than the standard relational model. By *generic*, we mean that some of the data structures described by the schema are not specific to the application domain the database is devoted to (e.g., columns Attribute and Value, which are domain independent). To better understand this concept, let us identify the objects of a standard relational schema the existence or the name of which is specific to the application domain, therefore providing information on this domain:

- **database**: the name of the database usually suggests the nature of the application domain (e.g., **ORDERS.db**)
- **table**: each table of a normalized database contains the data of a set of entities of the same type. The name of the table should be close to that of this entity type.
- **column**: each column designates a property of the entities described by its table. Here again, the name of the column should suggest that of this property.
- **data type**: some data types give hints on the meaning of the values of a column (e.g., date, character, integer and the user-defined domains)
- **unique key**: defines an integrity constraint that translates an important rule of the application domain (just think of the primary key (OrdID,ProdID) of table DETAIL).
- **foreign key**: generally defines relationships between rows, and, therefore, between entities.

Actually, the databases implemented in each of the alternate data models express (almost) the same types of facts of the application domain. They differ on how and where these fact types are specified: in the schema or in the data. For example, in the *Key-Value* model, *version 1*, the table name identifies an entity type but its attributes are defined by the value set of column **Attribute**.

This comparison is summarized in Table 6.1 below. The answer is **S** if the schema provides the information, **D** if the information is given by the data, **s** if the information is incompletely specified in the schema, **d** if the information is incompletely specified in the data and ∅ if the information is not specified.

Considering the standard relational model as the reference, this table evaluates to what extent the information contents of the schema of each alternative model is preserved (we ignore the database name, which is present in each model). Another way of describing this analysis could be: *what do I learn on the application domain when I read the schema*? And, as a consequence, *do the data include the missing information*?

Clearly, the schema of alternate data models is less rich and less expressive than the relational model. This weakness is compensated by a greater flexibility: new entity types and new attributes can be dynamically added without modifying the schema.

Table 6.1 - Data/metadata distribution in various data models

	Entity type	Property	Datatype	Unique key	Relationship
Relational	**S**	**S**	**S**	**S**	**S**
Universal table	**D**	**S**	**S**	∅	∅
Column tables	**s**[a]	**S**	**S**	**s**[b]	**s**[c]
Key-value 1	**S**	**D**	∅	∅	∅
Object-oriented	**S**	**D, s**[d]	∅	∅	∅
Key-value 2	**D**	**D**	∅	∅	∅

a.Maybe, through some parts of the name of the column table
b.Possible for single-component unique keys only
c.Possible for single-component foreign keys only
d.Maybe, through the name of the aggregate

6.5 For once, most *Object data models* are created equal

The data structures described in this study have been adopted by the most recent (and most popular) NoSQL DBMS, often under the family name *Document-oriented* DBMS. Indeed, hierarchical structures of arbitrary complexity are typical of documents and forms. This model nicely mimics the organization of a document, much more than flat relational structures, in which the components of documents are scattered among several tables. As an example, the data of customer orders are distributed in tables CUSTORDER and DETAIL of database ORDERS.db, while, in the object models, they are assembled into significant aggregates.

Among the most representative document-oriented DBMS we can mention MongoDB (the most popular in 2017) but also CouchDB, Azure, Datastore and Oracle NoSQL.

Interestingly, these hierarchical data structures which are at the basis of these modern DBMS, are very ancient. First, they are part of to most standard programming languages, such as COBOL (in which so-called *record structures* are particularly sophisticated), Pascal, C and, to some extent, Python.

A well known proverb teaches us that there is no free lunch. While the ultimate desirable form of relational databases comprises normalized tables[3], that are (1) devoid from any internal redundancies and (2) neutral *wrt* data application requirements, object databases generally include duplicated data and often are dedicated to a specific application (and therefore less easily reusable). In addition, the manipula-

3. The concept of relational normalization will be studied in a further case study.

tion languages of object data structures are more complex and less powerful than SQL.

6.6 The scripts

The scripts of the *Object* models as well as those of the models studied in Parts 1 and 2 are available in directory **SQLfast/Scripts/Case-Studies/Case_Schemaless**. They can be run from main script **Schemaless-MAIN.sql** (Figure 6.7).

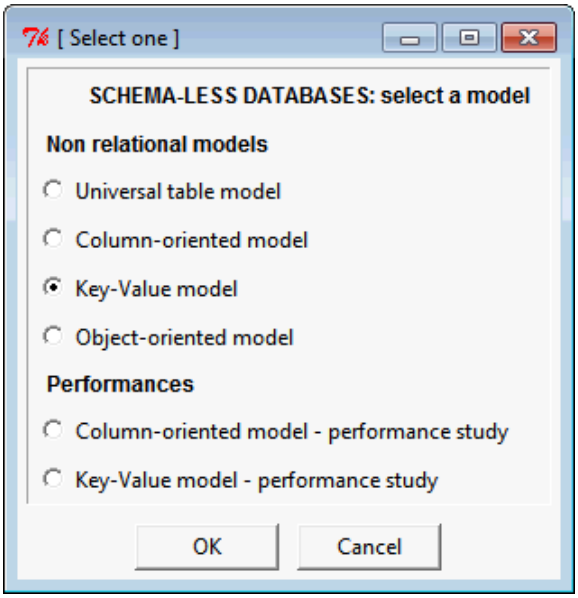

Figure 6.7 - Selecting a data model and experimenting with performances

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.