Case study 5

Schema-less databases - Part 2

Objective: This document studies a third family of alternative data models, namely the *Key-Value* data structure. In these models, the information is represented by triples that each defines the value of an attribute of an entity. Several approaches are described, with increasing levels of genericity. As in Part 1, migration and exploitation scripts are developed for the ORDERS.db database.

Keywords: non-relational data model, NoSQL, schema-less database, key-value model, triple, triplestore, RDF, SPARQL, description logic, A-BOX, OWL, Redis, Berkley DB, data migration, schema conversion

5.1 Introduction

The *Key-Value model* is at the basis of many NoSQL DBMS (though we will see that this name is misleading) but also of knowledge representation formalisms such as OWL. The basic idea is the following: instead of storing value FERARD into column **Name** of the CUSTOMER row describing customer C400, we store the triple (C400, Name, FERARD) into a 3-column table named, say, **M_CUSTOMER**.

A representative of this family of data model has been studied in Chapter 16 of the SQLfast tutorial (*Key-Value output format*), in which the data of a table are transformed into the format key = value through style script UTIL-SELECT-parameters-for-KEY-VALUE:

```
CustID = B112
Name = HANSENNE
Address = 23, r. Dumont
etc.
```

In this section we will examine several ways to exploit the *Key-value* format, in which some schema information (metadata), such as *column names*, are moved to user data: Name is no longer the name of a column of the schema but just plain user data.

The term *Key* being too technical for the domain we will discuss, we will use the term ***attribute*** instead (**Name** is not a *key* of customers but an *attribute* or a property thereof). However, we will keep the name *Key-Value* to designate this family of models, a term which has become standard.

5.2 The Key-Value model - Version 1

The first approach consists in storing the data about each entity type in a specific table. So, we create tables **M_CUSTOMER**, **M_CUSTORDER**, **M_DETAIL** and **M_PRODUCT**, each containing the data of the corresponding table of the source database.

Each table comprises three columns:

- **Entity**, which denotes an entity of the application domain (a customer, an order, a detail, a product), either through some kind of primary key (CustID for customers for example) or through an abstract, technical identifier such as a simple integer,
- **Attribute**, which contains the name of an attribute (e.g., Name, City, DateOrd, Price),
- **Value**, which contains the value of this attribute for this entity (e.g., Férard, London, 2017-03-28, 123.5).

For instance, row (C400, Name, FERARD) of table **M_CUSTOMER** tells that the customer denoted by **C400** has an attribute called **Name**, the value of which is **FERARD**. Such rows are called *triples*. Excerpts of this table, which comprises 16 x 6 = **96** triples, are shown in Figure 5.1.[1]

```
+--------+-----------+----------------------+
| Entity | Attribute | Value                |
+--------+-----------+----------------------+
| B062   | CustID    | B062                 |
| B062   | Name      | GOFFIN               |
| B062   | Address   | 72, r. de la Gare    |
```

1. The way *null* values are displayed in the result of `select` statements (here `--`) is specified by parameter dispNull in initialization file SQLfast.ini.

```
| B062     | Account    | -3200.0                |
| B062     | Cat        | B2                     |
| B062     | City       | Namur                  |
| B112     | CustID     | B112                   |
| B112     | Name       | HANSENNE               |
| B112     | Address    | 23, r. Dumont          |
| B112     | City       | Poitiers               |
| B112     | Cat        | C1                     |
| B112     | Account    | 1250.0                 |
| ...      | ...        | ...                    |
| C400     | CustID     | C400                   |
| C400     | Name       | FERARD                 |
| C400     | Address    | 65, r. du Tertre       |
| C400     | City       | Poitiers               |
| C400     | Cat        | B2                     |
| C400     | Account    | 350.0                  |
| D063     | CustID     | D063                   |
| D063     | Name       | MERCIER                |
| D063     | Address    | 201, bvd du Nord       |
| D063     | City       | Toulouse               |
| D063     | Cat        | --                     |
| D063     | Account    | -2250.0                |
| ...      | ...        | ...                    |
+--------+-----------+----------------------+
```

Figure 5.1 - Excerpts of table M_CUSTOMER

5.2.1 Generating Key-Value data from the source tables

Script 5.1 creates the first two tables, M_CUSTOMER and M_CUSTORDER. We notice that, apart from their names, these tables have exactly the same definition. We also observe that column Value has been declared **`varchar`**, whatever the type, numeric or character, of the source values that we will store in it.

Loading the data from the source database into the Key-value tables is carried out by Script 5.2, which is simple and intuitive, but may not be very efficient for large tables with many columns since it parses table CUSTOMER once for each of its columns.

Script 5.3 suggests an alternative technique to migrate the source data. It parses source table CUSTOMER only once. For each source row and for each source column, it inserts an attribute and its value in target table M_CUSTOMER.

The fact that source values are first stored in SQLfast variables `cus`, `nam`, `add`, `cit`, `cat` and `acc` before being inserted as constants in `insert` queries requires the use of special variable-value substitution delimiter **§** instead of **$**. This way, internal single quotes are doubled in character string constants, as required by the SQL syntax.

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

Script 5.1 - Structure of tables M_CUSTOMER and M_CUSTORDER

```
insert into M_CUSTOMER (Entity,Attribute,Value)
       select CustID, 'CustID', CustID from S_CUSTOMER;
 union select CustID, 'Name', Name from S_CUSTOMER;
 union select CustID, 'Address', Address from S_CUSTOMER;
 union select CustID, 'City', City from S_CUSTOMER;
 union select CustID, 'Cat', Cat from S_CUSTOMER;
 union select CustID, 'Account',
              cast(Account as char) from S_CUSTOMER;
```

Script 5.2 - Generation of the contents of table M_CUSTOMER (pure SQL version)

```
for cus,nam,add,cit,cat,acc = [select CustID,Name,Address,City,
                               Cat,Account from CUSTOMER]
   insert into M_CUSTOMER values
          ('§cus§','CustID','§cus§'),
          ('§cus§','Name','§nam§'),
          ('§cus§','Address',§add§),
          ('§cus§','City','§cit§'),
          ('§cus§','Cat',
           case when ('§cat§' = '') then null else '§cat§' end),
          ('§cus§','Account',cast($acc$ as char));
```

Script 5.3 - Generation of the contents of table M_CUSTOMER (procedural version)

Observing that column **Cat** is *nullable*, we have to decide the way *null* values will be represented: either by an attribute triple with a *null* value or by discarding this triple for this entity. As suggested in Figure 5.1, we choose the first approach. Since the `SQL-for` loop returns an empty string for each *null* value, we must translate this

empty string into a *null* value before inserting the triple in M_CUSTOMER. The SQL conversion expression is simple:

```
case when '§cat§' = '' then null else '§cat§' end
```

We also note that the values of column Account, being numeric, must be converted in character strings, which is required by the type of column Value.

5.2.2 The case of composite primary keys

All this works fine for source tables the primary key of which comprises one column only. The case of table DETAIL is a bit more complex. The first idea would be to create a Key-value table with four columns (Script XXX).

```
create table M_DETAIL(
   Entity1   varchar(32) not null,
   Entity2   varchar(32) not null,
   Attribute varchar(32) not null,
   Value     varchar(32),
   primary key (Entity1,Entity2,Attribute));
```

Script 5.4 - M_DETAIL: strange triples with four components!

This seems a bad idea in that it produces tables with different schemas, therefore spoiling the nice uniform data structures. So, we better try to preserve the uniformity of the concept of entity Id. Two valid techniques:

1. Denoting entities with an abstract identifier, such a pure integer, as we did in some techniques of Part 1.
2. Building a single-valued id from the components of the primary key. For table DETAIL, considering the syntax of OrdID and ProdID, their concatenation would be fine:

   ```
   OrdID||'-'||ProdID
   ```

With the second technique, the result is quite satisfying (Figure 5.2).

Entity	Attribute	Value
30178-CS464	OrdID	30178
30178-CS464	ProdID	CS464
30178-CS464	Qord	25
30179-CS262	OrdID	30179
30179-CS262	ProdID	CS262
30179-CS262	Qord	60
...	...	...
30188-PH222	OrdID	30188

```
| 30188-PH222 | ProdID     | PH222 |
| 30188-PH222 | Qord       | 92    |
+-------------+-----------+-------+
```

Figure 5.2 - Building triples from a table with a composite primary key

5.2.3 Rebuilding the source tables from their Key-Value expression

Script 5.5 rebuilds source table CUSTOMER by a multiple self-join of table M_CUSTOMER.

```
insert into TMP_CUSTOMER(CustID,Name,Address,City,Cat,Account)
select M1.Value as CustID, M2.Value as Name,
       M3.Value as Address,
       M4.Value as City, M5.Value as Cat,
       cast(M6.Value as decimal) AS Account
from   M_CUSTOMER M1, M_CUSTOMER M2, M_CUSTOMER M3,
       M_CUSTOMER M4, M_CUSTOMER M5, M_CUSTOMER M6
where  M2.Entity = M1.Entity and M3.Entity = M1.Entity
and    M4.Entity = M1.Entity and M5.Entity = M1.Entity
and    M6.Entity = M1.Entity
and    M1.Attribute = 'CustID'
and    M2.Attribute = 'Name'
and    M3.Attribute = 'Address'
and    M4.Attribute = 'City'
and    M5.Attribute = 'Cat'
and    M6.Attribute = 'Account'
order by CustID;
```

Script 5.5 - Generation of the contents of table CUSTOMER (SQL version 1)

Script 5.6, though a bit less intuitive at first glance, should be better, since it (hopefully) scans the table only once. It forms a group of triples for each entity. In each group, it extracts the highest value of each attribute. Among the triples of this group, only one has a true value for this attribute while all the others are assigned value `''`:

```
max(case when Attribute = 'Name' then Value else '' end)
```

Script 5.7, based on a simple scan of the source table, exhibits some interesting programming patterns. The rows are sorted by **Entity** values, in such a way that all the attribute rows of each entity are read consecutively. The **`extract`** statement starts the procedure by getting the first customer Id. Then, a loop parses all the rows of M_CUSTOMER. Variable `ce` denotes the row currently being built. The successive rows with the same Entity value form one row of the **TMP_CUSTOMER** table (a

clone of table CUSTOMER). The result of reading these rows is stored in variables `cus`, `nam`, `add`, `cit`, `cat` and `acc`, that are then used to insert the current TMP_CUSTOMER row.

When the loop terminates, these variables contain the data of the last row that has still to be inserted.

The algorithm of this script is the usual way that has been used for decades to create and process groups of records in sequential file processing. This approach is interesting from a historical point of view (it may also help to solve some special intricate problems) but it prevents the SQL engine to optimize the generation of triples.

.

```
insert into TMP_CUSTOMER(CustID,Name,Address,City,Cat,Account)
max(case when Attribute = 'CustID'
         then Value else '' end) as CustID,
max(case when Attribute = 'Name'
         then Value else '' end) as Name,
max(case when Attribute = 'Address'
         then Value else '' end) as Address,
max(case when Attribute = 'City'
         then Value else '' end) as City,
max(case when Attribute = 'Cat'
         then Value else null end) as Cat,
max(case when Attribute = 'Account'
         then cast(Value as decimal) else -9999.9 end) as
Account
from  M_CUSTOMER group by Entity;
```

Script 5.6 - Generation of the contents of table CUSTOMER (SQL version 2)

5.2.4 About performance

Figure 5.3 compares the execution time of each variant of the two conversion processes for increasing size of table CUSTOMER. These figures have been computed by the script **Schemaless2-Key-Value-Performance.sql** in directory **Case_Schemaless**.

- *Generating column-tables*: the pure SQL formula, that convert 32,768 source rows in 0.8 second, is much faster than the procedural technique that requires nearly one minute. However, the latter figure is not really significant. The procedure has been written in SQLfast, whose execution overhead is (much) higher than that of standard languages such as C, Java or even Python.
- *Rebuilding CUSTOMER table*: the multi-join SQL is the best technique to recover the source table: it converts about 200,000 triples into 32,768 source rows in less than 0.33 s. The SQL single scan is also quite good, with an execu-

tion time of less than 0.45 s. The procedural technique has soon lost the race, for the same reason we have cited above.

Nevertheless, our goal merely is to experiment the concepts of alternative models and not to develop high performance scripts for multi-terabyte databases!

```
extract ce = select Entity from M_CUSTOMER order by Entity;
for e,a,v = [select * from M_CUSTOMER order by Entity];
   if ('$e$' = '$ce$');
      if ('$a$' = 'CustID')  set cus = $v$;
      if ('$a$' = 'Name')    set nam = $v$;
      if ('$a$' = 'Address') set add = $v$;
      if ('$a$' = 'City')    set cit = $v$;
      if ('$a$' = 'Cat')     set cat = $v$;
      if ('$a$' = 'Account') set acc = $v$;
   else;
      insert into TMP_CUSTOMER values('§cus§','§nam§','§add§',
      '§cit§',case when '§cat§'='' then null else '§cat§' end,
       cast($acc$ as decimal));
      set ce = $e$;
      if ('$a$' = 'CustID')  set cus = $v$;
      if ('$a$' = 'Name')    set nam = $v$;
      if ('$a$' = 'Address') set add = $v$;
      if ('$a$' = 'City')    set cit = $v$;
      if ('$a$' = 'Cat')     set cat = $v$;
      if ('$a$' = 'Account') set acc = $v$;
   endif;
endfor;
insert into TMP_CUSTOMER values('§cus§','§nam§','§add§',
      '§cit§',case when '§cat§'='' then null else '§cat§' end,
       cast($acc$ as decimal));
```

Script 5.7 - Generation of the contents of table CUSTOMER (procedural version)

Size	16	256	1,024	4,096	32,768
Load M_CUSTOMER (SQL version)	1	4	23	81	829
Load M_CUSTOMER (procedural)	17	264	1,146	5,046	56,978
Rebuild CUSTOMER (SQL 1)	1	2	10	34	321
Rebuild CUSTOMER (SQL 2)	1	3	11	47	445
Rebuild CUSTOMER (Procedural)	307	5,395	23,313		

Figure 5.3 - Execution time, in ms., of the conversion processes applied to source CUSTOMER table for various number of rows

5.2.5 Application

The *Key-value* model allows the same queries to be formulated as those working on the source database. Scripts 5.8 to 5.9 are some examples.

```
select C1.Entity as CustID,C1.Value as Name,C2.Value as City
from   M_CUSTOMER C1, M_CUSTOMER C2
where  C1.Entity = C2.Entity
and    C1.Attribute = 'Name'
and    C2.Attribute = 'City'
order by C1.Entity;
```

Script 5.8 - Extracting data CustID, Name and City of the rows of table M_CUSTOMER

```
select Entity as CustID, Value as Name
from   M_CUSTOMER
where  Attribute = 'Name';
```

Script 5.9 - Extracting data CustID and Name from the rows of table M_CUSTOMER

However, new kinds of queries can be written, based on the fact that the values of different attributes are stored in the same column (Script 5.10) and on the fact that attribute names are pure values which can be queried as well (Script 5.11)

```
select Entity,Attribute,Value
from   M_CUSTOMER
where  Value like '%B1%'
order by Entity;
```

Script 5.10 - Which customers have a property value including characters *B1*?

```
select Entity,Attribute,Value
from   M_SHIPMENT
where  lower(Attribute) like '%date%'
order by Entity;
```

Script 5.11 - Extracting all the data of table M_SHIPMENT where some attribute names include the word **date** (whatever the case), such as *ShipmentDate* and *PaymentDate*

5.3 The Key-Value model - Version 2

In this variant of the *Key-Value* model, all the data of a database are stored in a **single table**, whatever the entity type they belong to. Thus, there is one table for the whole source database. The three columns **Entity**, **Attribute** and **Value** still are valid, but a fourth column, **EType** is necessary if we want to keep the notion of *entity type*. Considering the source ORDERS.db database, it is natural to call this unique table **ORDERS**. This table can be declared by Script 5.12.

```
create table ORDERS(
   EType      varchar(18) not null,
   Entity     varchar(15) not null,
   Attribute  varchar(32) not null,
   Value      varchar(64),
   primary key (EType,Entity,Attribute));
```

Script 5.12 - Table ORDERS hosts all the data of database ORDERS.

Figure 5.4 is an excerpt of the contents of table ORDERS. The entity ID of an entity is the primary key value of the source row.

EType	Entity	Attribute	Value
CUSTOMER	B112	CustID	B112
CUSTOMER	B112	Name	HANSENNE
CUSTOMER	B112	Address	23, r. Dumont
CUSTOMER	B112	City	Poitiers
CUSTOMER	B112	Cat	C1
CUSTOMER	B112	Account	1250
...	...	...	...
PRODUCT	CS262	ProdID	CS262
PRODUCT	CS262	Description	RAFT. PINE 200x6x2
PRODUCT	CS262	Price	75
PRODUCT	CS262	QonHand	45
...	...	...	...
CUSTORDER	30178	OrdID	30178
CUSTORDER	30178	CustID	K111
CUSTORDER	30178	DateOrd	2013-12-21
...	...	...	...
DETAIL	30178-CS464	OrdID	30178
DETAIL	30178-CS464	ProdID	CS464
DETAIL	30178-CS464	Qord	25
...	...	...	...

Figure 5.4 - Contents of table ORDERS

Rebuilding the CUSTOMER table from ORDERS is a problem similar to that addressed in Section 5.2.3. Script 5.13 derives from variant 2 of Script 5.6.

```
select
max(case when Attribute = 'CustID'
         then Value else '' end) as CustID,
max(case when Attribute = 'Name'
         then Value else '' end) as Name,
max(case when Attribute = 'Address'
         then Value else '' end) as Address,
max(case when Attribute = 'City'
         then Value else '' end) as City,
max(case when Attribute = 'Cat'
         then Value else null end) as Cat,
max(case when Attribute = 'Account'
         then cast(Value as decimal) else -9999.9 end)
         as Account
from   ORDERS
where  EType = 'CUSTOMER'
group by Entity;
```

Script 5.13 - Reconstruction of table CUSTOMER from table ORDERS (variant 2)

5.4 The Key-Value model - Version 3

We can keep the triple format by merging columns **EType** and **Entity**, in such a way that the values of this column are unique among all the entities of the application domain (Figure 5.5). In this approach, the type of an entity is just an additional attribute of the latter. For example, customer entity **B112** can be uniquely denoted by **CUSTOMER-B112**. Its type is '**CUSTOMER**', while its name is '**HANSENNE**'.

By using abstract entity Ids we can shrink column Entity and derive a more concise triple table. However, the generation of these entity Ids is a bit more complex than it was in the context of a single table. Here, all the entities of the database are assigned a *globally unique* Id, whatever their type. This can be done in several ways. One of them consist in creating an *Id dictionary* table in which, before inserting a new entity, we record its type and primary key (column EType and Entity as in Figure 5.4), so that a global abstract id, declared *autoincrement*, is automatically generated. The latter is then used to denote this entity in table ORDERS.

A trade-off between these entity identification techniques consists in coding both the entity type and the entity within its type. This requires a small dictionary table that assigns to each table name a numeric code and records the *last entity number* created for this table. Before inserting an entity, we extract the table code from this dictionary and we increment the last entity number. Figure 5.6 shows excerpts of table ORDERS where column Entity denotes entities with this coding technique.

Figure 5.7 suggests a graphical interpretation of the composition of CUSTOMER entity 01-00012.

Entity	Attribute	Value
CUSTOMER-B112	EType	CUSTOMER
CUSTOMER-B112	CustID	B112
CUSTOMER-B112	Name	HANSENNE
CUSTOMER-B112	Address	23, r. Dumont
CUSTOMER-B112	City	Poitiers
CUSTOMER-B112	Cat	C1
CUSTOMER-B112	Account	1250
...	...	...
PRODUCT-CS262	EType	PRODUCT
PRODUCT-CS262	ProdID	CS262
PRODUCT-CS262	Description	RAFT. PINE 200x6x2
PRODUCT-CS262	Price	75
PRODUCT-CS262	QonHand	45
...	...	...

Figure 5.5 - Contents of database ORDERS reduced to triples (1)

Entity	Attribute	Value
01-001	EType	CUSTOMER
01-001	CustID	B112
01-001	Name	HANSENNE
01-001	Address	23, r. Dumont
01-001	City	Poitiers
01-001	Cat	C1
01-001	Account	1250
...	...	...
04-001	EType	PRODUCT
04-001	ProdID	CS262
04-001	Description	RAFT. PINE 200x6x2
04-001	Price	75
04-001	QonHand	45
...	...	...

Figure 5.6 - Contents of database ORDERS reduced to triples (2)

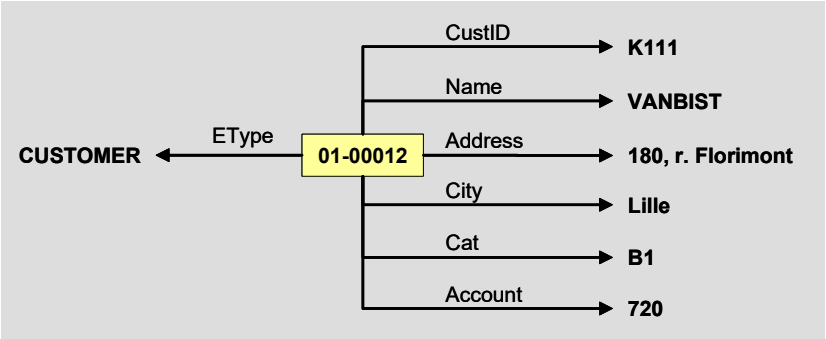

Figure 5.7 - Graphical representation of CUSTOMER entity 01-00012

It is important to observe that the entity Ids in column Entity, despite the way they are built, basically are **meaningless**. They could be replaced by arbitrary numbers or character strings without information loss.

The generation of table ORDERS from the tables of database ORDERS.db is left as an exercise.

This flexible way to represent facts about an application domain has been used to support several popular knowledge representation languages and systems. RDF (Resource Description Framework) is one of them.[2] It represents facts through a graph of resources. A resource is an *object* of any kind available locally or, more generally, on the web. Resources are identified by their URI (Universal resource identifier) or by a simple literal (number or character string). HTTP URL are a special kind of URI. Edges between two resources specifies that the first one (the *subject*) has, as *property*, the second one (the *object*).

RDF data generally are stored in triplestore[3], databases optimized to efficiently store, manage and process large triplestores (currently several trillions triples).[4] A graph query language, SPARQL, derived from SQL, is associated with this data model.

5.4.1 Application to Description Logic

Description Logic (DL) is a formal language derived from *First-order Logic* (FOL) and intended to precisely represent knowledge about a certain *universe* (what we have called the *application domain*) and to reason on it. DL exists in several variants, depending on the part of FOL they translate. DL is used to define facts about the universe, just like a database does. In addition, it allows general rules to be stated, either to specify those of these facts that are valid (kind of *integrity constraints*) and to infer new facts from those we already know.[5]

DL is the formal basis of ontology languages, in particular for the Semantic Web, such as DAML, OIL, OWL and, more recently OWL2.

Several syntax have been proposed and inference engines have been developed to define, store and process facts and general rules. The variant of DL we will talk about in this section is particular simple, since it addresses the representation of facts only. It comprises three constructs:

– *individual*: constant that is intended to denote a concrete or abstract thing of the universe. For instance, 01-00001 and Toulouse are individuals.

2. https://www.w3.org/RDF/
3. https://en.wikipedia.org/wiki/Triplestore
4. https://www.w3.org/wiki/LargeTripleStores
5. https://en.wikipedia.org/wiki/Description_logic [July 2017]; Baader, F., Horrocks, C., Sattler, U., *An Introduction to Description Logics*, Cambridge University Press, 2017

– *concept*: unary predicates that assigns a class to individuals. CUSTOMER(01-00001) asserts that individual 01-00001 belongs to class CUSTOMER and STRING(Toulouse) tells that individual Toulouse belongs to class STRING.

– *roles*: binary predicates used to assert a relationship between two individuals. City(01-00001,Toulouse) tells that Toulouse is the (name of) city of customer (denoted by) 01-00001.

The instantiation of concepts and roles with individuals forms a set of assertions that describe the universe. In the DL vocabulary, they are collectively called the A-BOX.

Actually, table ORDERS, that we have generated in this version of the Key-value model, contains everything necessary to generate the A-BOX of the DL description of our small universe comprising *customers*, *orders*, *details* and *products*. Generating the concept and role assertions requires two simple SQL queries (Scripts 5.14 and 5.15).

```
select distinct Value||'('''||Entity||''')' as Individuals
from   ORDERS
where  Attribute = 'EType';
```

Script 5.14 - Generating concept assertions from table ORDERS

```
select Attribute||'('||''''||Entity||''','''
       ||coalesce(doubleQuote(Value),'')||''')' as Roles
from   ORDERS
where  Attribute <> 'EType';
```

Script 5.15 - Generating role assertions from table ORDERS

Excerpts of their result sets is shown in Figures 5.8 and 5.9. Some comments on these queries:

– the individual notations are quoted to allow commas inside values

– consequently, internal quotes must be doubled. This is the role of UDF function **doubleQuote**.

– **coalesce** is a standard SQL function that returns the first non null value within its argument list.

```
CUSTOMER('01-00001')
CUSTOMER('01-00002')
CUSTOMER('01-00003')
...
PRODUCT('02-00001')
PRODUCT('02-00002')
...
CUSTORDER('03-00001')
CUSTORDER('03-00002')
```

```
...
DETAIL('04-00001')
DETAIL('04-00002')
...
```

Figure 5.8 - Excerpts of the assertions of the A-BOX (the individuals and the concepts)

```
CustID('01-00001','B062')
Name('01-00001','GOFFIN')
Address('01-00001','72, r. de la Gare')
City('01-00001','Namur')
Cat('01-00001','B2')
Account('01-00001','-3200')
...
CustID('01-00010','F011')
Name('01-00010','PONCELET')
Address('01-00010','17, Clôs des Erables')
City('01-00010','Toulouse')
Cat('01-00010','B2')
Account('01-00010','0')
...
OrdID('03-00005','30185')
CustID('03-00005','F011')
DateOrd('03-00005','2016-01-02')
...
OrdID('04-00007','30185')
ProdID('04-00007','CS464')
Qord('04-00007','260')

OrdID('04-00008','30185')
ProdID('04-00008','PA60')
Qord('04-00008','15')

OrdID('04-00009','30185')
ProdID('04-00009','PS222')
Qord('04-00009','600')
...
ProdID('02-00005','PA60')
Description('02-00005','NAILS STEEL 60 (1K)')
Price('02-00005','95')
QonHand('02-00005','134')
...
ProdID('02-00007','PS222')
Description('02-00007','PL. PINE 200x20x2')
Price('02-00007','185')
QonHand('02-00007','1220')
...
ProdID('02-00003','CS464')
Description('02-00003','RAFT. PINE 400x6x4')
Price('02-00003','220')
QonHand('02-00003','450')
...
```

Figure 5.9 - Excerpts of the assertions of the A-BOX (the roles)

In a relational database, a relationship between two entities is expressed by a *foreign keys* between the rows that describe these entities. If the *foreign key value* of the

source row is equal to the *primary key value* of the target row, then a relationship exists between these rows, and, as a consequence, between the entities described by these rows. This complicated way to state inter-entity relationship becomes much more simple in Description logic.

For example, the fact that the **order** denoted by **`x`** (whatever its CustID value) has been placed by the **customer** denoted by **`y`** (whatever their CustID value) will be simply stated by the assertion:

```
placed_by(x,y)
```

In this representation, the assertions that represent the columns of foreign keys become useless and can be discarded.

Figure 5.10 collects the assertions that express the relationships between orders and customers. The first expression tells us that the order denoted by `'03-00001'` has been placed by the customer denoted by `'01-00012'`.

```
placed_by('03-00001','01-00012')
placed_by('03-00002','01-00007')
placed_by('03-00003','01-00015')
placed_by('03-00004','01-00007')
placed_by('03-00005','01-00010')
placed_by('03-00006','01-00007')
placed_by('03-00007','01-00004')
```

Figure 5.10 - Derivation of *placed_by* role assertions

Actually, `placed_by` assertions can be derived from the assertions of Figure 5.9 through a conversion process illustrated in Figure 5.11.

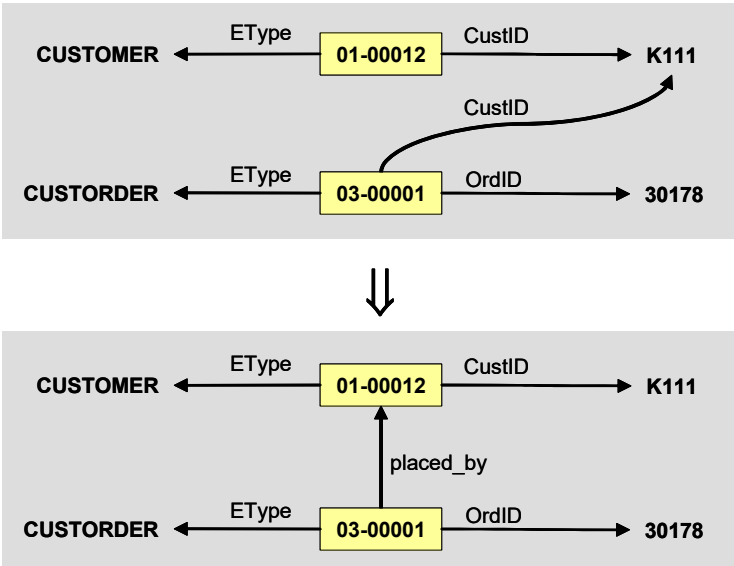

Figure 5.11 - Production of assertions of role *placed_by*

Since the description of the rule language of DL is outside the scope of this study, we will express this derivation rule as an equivalent[6] SQL query on the triples of Figure 5.5.

To help building this query, we will use the example of order 30178 that has been placed by customer K111. The triples describing these two entities are reminded in Figure 5.12. The four triples that will be used are marked with tags `O1`, `O2`, `O3`, `O4`. The reasoning is as follows:

- the order is identified by three properties:
  - it is of type CUSTORDER [`O1.Attribute = 'EType' and O1.Value = 'CUSTORDER'`]
  - the value of its attribute CustID is **V** [`O2.Attribute = 'CustID'`]
  - its entity Id is **W** in the two triples [`O2.Entity = O1.Entity`]
- the customer is identified by three properties:
  - it is of type CUSTOMER [`O3.Attribute = 'EType' and O3.Value = 'CUSTOMER'`]
  - the value of its attribute CustID is **X** [`O4.Attribute = 'CustID'`]
  - its entity Id is **Y** in the two triples [`O4.Entity = O3.Entity`]
- the link between them is expressed by property:
  - **V = X** [`O2.Value = O4.Value`]
- the role assertion can then be built:
  - `placed_by(W,Y)` [`O1.Entity`, `O3.Entity`]

The derivation query is shown in Script 5.16.

Entity	Attribute	Value	
01-00012	**EType**	**CUSTOMER**	**O3**
01-00012	**CustID**	**K111**	**O4**
01-00012	Name	VANBIST	
01-00012	Address	180, r. Florimont	
01-00012	City	Lille	
01-00012	Cat	B1	
01-00012	Account	720	
...	...	...	
03-00001	**EType**	**CUSTORDER**	**O1**
03-00001	OrdID	30178	
03-00001	**CustID**	**K111**	**O2**
03-00001	DateOrd	2015-12-21	
...	...	...	

Figure 5.12 - Two sets of roles expressing two entities and their relationship

6. Triples and role assertions are equivalent and both SQL and DL are based on FOL

```
select 'placed_by('''||O1.Entity||''','''||O3.Entity||''')'
from  ORDERS O1, ORDERS O2, ORDERS O3, ORDERS O4
where O1.Attribute = 'EType'  and O1.Value = 'CUSTORDER'
and   O2.Attribute = 'CustID' and O2.Entity = O1.Entity
and   O3.Attribute = 'EType'  and O3.Value = 'CUSTOMER'
and   O4.Attribute = 'CustID' and O4.Entity = O3.Entity
and   O2.Value = O4.Value;
```

Script 5.16 - SQL query that create *placed_by* assertions

Finally, what are the differences between triples, RDF, OWL and DL?

All this may seem a bit complicated. Let us try to clarify (a little bit) the relationship between these concepts.[7]

– *Triple* is a data format. A *triplestore* is a specialized database in which triples can be stored and retrieved efficiently.

– *RDF* is a logical model and languages (including *SPARQL*) suited to interpret triples as pieces of knowledge independently of the underlying storage technology. Just like the relational model (including SQL) interprets the records stored in the files of a database.

– *OWL* can be seen as an extension of RDF based on the concept of *ontology*.[8]

– *Description logics* is a mathematical theory derived from the FOL, that gives RDF and OWL a sound formal interpretation. Just like the theoretical relational model of E. F. Codd gives RDBMS a sound basis to interpret the result of any query. In particular, DL provides mechanism to define inference rules.

5.5 Not all Key-value models are created equal

Some popular NoSQL DBMS are said to belong to the *Key-value* data manager family. Once again, this name is completely misleading. *Key-value* DBMS propose to organize data according to a low level model in which a unique key (the *Key*) is associated to (generally) unstructured chunks of data (the *Value*). These data can be retrieved by their *Key* value. Point!

7. A quite interesting discussion on this topic is available at https://stackoverflow.com/questions/1740341/what-is-the-difference-between-rdf-and-owl.
8. An *ontology* is a structured collection of terms that describe the *things* of a certain application domain in terms of entities, classes of entities and relationships between them. As compared with databases, an ontology is, to some extent, similar to the conceptual schema of a database (though an ontology may include instances as well).

Practically, The schema of any *Key-value* database could be simulated by a relational table:

```
create table RECORD(Key varchar(2048) not null primary key,
                    Value blob);
```

The structure and the meaning of the *Key* and *Value* parts of records are up to the application. There is no query language. The data manipulation primitives are reduced to **`get`**, **`put`** and **`delete`**. A *Key-value* database is just an abstract repository in which data of arbitrary complexity can be stored and retrieved by a unique key. The main role of these DBMS is the efficient management of large volumes of data massively distributed and replicated among many data servers. *Redis* and *Berkeley DB* are among the most popular.

Quite often, a *Key-value* data manager is the bottom layer of more sophisticated manager, such as the NoSQL DBMS provided by Oracle, that comprises *Berkeley DB* topped by a JSON upper layer.

5.6 The scripts

The scripts of the *Key-value* models are available in directory **SQLfast/Scripts/Case-Studies/Case_Schemaless**. They can be run from main script **Schemaless-MAIN.sql** (see Part 3)