

Case study 4

Schema-less databases - Part 1

Objective: This document is the first of a series of three case studies that explore alternative data models that organize the data in structures that provide more flexibility than standard relational tables. We observe that this flexibility makes it easier to dynamically modify the schema of the database but, as an unfortunate consequence, the schema is less expressive. In some of these data models, the schema is practically devoid of any information, hence the name *schema-less*.

In this study, we explore two extreme data models. In the *Universal table model*, according to which the data of all the source tables are stored in a single table comprising the union of the columns of the source tables. In the second model, called *Column-oriented model*, each column of the source tables is implemented in an independent table.

Scripts migrating the data of the ORDERS.db database to these models, and conversely, are developed.

Keywords: non-relational data model, NoSQL, schema-less database, universal table model, universal relation, column-oriented data model, Cassandra, data migration, schema conversion

4.1 Introduction

The organization of data in columns and rows stored in tables, the so-called *standard relational model*, is the most usual way to structure a database. Each table collects the information on a set of real world entities (customers, products, employees, movies, etc.). In a table, each entity is described by one and only one row and each column stores a definite property of these entities. The name of the table is close to the name of the entity set (CUSTOMER, EMPLOYEE) and the name of each column evokes a property (Name, Address, City, Price, Skill).

However, this natural way to organize data is not the only one. When performance and scalability become premium criteria, current relational database managers may prove inflexible to cope with high volume of rapidly evolving and growing data.

This is where the so-called **NoSQL** data managers begin to enter the scene. Their architecture favors scalability (they allow data to grow several orders of magnitude without administrative burden nor loss of performance), distribution, redundancy and flexible data structures. However, there is a price to pay: practically no centralized control of data quality, no high-level query language, no standard data structures and interface, no strong transactions. Most often these databases have no expressive data structures, in such a way that data integrity, quality control, semantic validation and the like are the responsibility of the application programs.

This document and the next two ones are a short but practical introduction to some of these alternative data models. Instead of working with native NoSQL DBMS (which one to choose among the nearly hundred of them?), we will express these models in relational data structures, develop scripts to migrate *source tables* (the four tables of database ORDERS.db) into these new models (and back) and write some queries for these data.

A last word: the title of these case studies is standard but misleading. Each database, whatever its model, *has a schema* but this schema can be more or less abstract and generic. *Schema-less* qualifies databases the structure of which tends to be simplistic, unspecific and merely technical. In other words, the names of objects of the schema (tables and columns) no longer denote entities and properties of the real world.¹

4.2 The Universal table model

This first alternative data model organizes the data of a database in a **single table** comprising one *not null* column (the table *primary key*) and a series of *nullable*

1. “The absence of a schema has some flexibility advantages, although for querying the data, the absence of a schema presents some challenges to people accustomed to a classic RDBMS.”—Iran Hutchinson. <http://www.odbm.org/blog/2014/01/big-data-three-questions-to-intersystems/>

columns. Each entity of the application domain (i.e., the part of the real world about which we want to record information) is described by a row where the relevant columns have been assigned a value while all the other ones are set to *null*. If needed, an additional column, let us call it **EType**, can be added to identify the type of the entity. These rows are assigned an abstract technical Id that act as a global primary key, independently of the primary keys of the source tables.

Script 4.1 shows an example of creation query for this table, called **DATA**. The abstract Id is called **Entity** since it uniquely denotes each entity of the application domain, whatever its type.

```
create table DATA(
  Entity      integer not null primary key autoincrement,
  EType       char(18) not null,
  CustID      char(10),
  Name        char(32),
  Address     char(60),
  City        char(30),
  Cat         char(2),
  Account     decimal(9,2),
  ProdID      char(15),
  Description  char(60),
  Price       integer,
  QonHand     integer,
  OrdID       char(12),
  DateOrd     date,
  Qord        integer);
```

Script 4.1 - Table DATA collecting rows of different patterns (from the standard ORDERS database) in a single table.

4.2.1 Generating the Universal table data from source tables

Script 4.2 migrates the data of database ORDERS and Figure 4.1 shows the contents of table DATA. The fact that most columns are set to *null* is less of a problem than it seems to be: in many DBMS implementations, a *null* value just costs one bit.

When possible, the same column is used to represent different roles according to the entity described. It is the case for CustID, OrdID and ProdID, that each appear in two entity types, once as a primary key and once as a foreign key.

4.2.2 From the Universal table back to the standard tables

Recovering source data from universal table DATA is straightforward. Script 4.3 declares a set of views that present the data according to the standard source tables.

```

insert into DATA (EType, CustID, Name, Address, City, Cat, Account)
select 'CUST', CustID, Name, Address, City, Cat, Account from CUSTOMER;
insert into DATA (EType, ProdID, Description, Price, QonHand)
select 'PROD', ProdID, Description, Price, QonHand from PRODUCT;
insert into DATA (EType, OrdID, CustID, DateOrd)
select 'ORD', OrdID, CustID, DateOrd from CUSTORDER;
insert into DATA (EType, OrdID, ProdID, Qord)
select 'DET', OrdID, ProdID, Qord from DETAIL;

```

Script 4.2 - Migration of the ORDERS database to table DATA

```

create view CUSTOMER as
select  CustID, Name, Address, City, Cat, Account
from    DATA where EType = 'CUST';

create view CUSTORDER (OrdID, CustID, DateOrd) as
select  OrdID, CustID, DateOrd
from    DATA where EType = 'ORD';
. . .

```

Script 4.3 - Rebuilding the tables of the ORDERS database through views defined on table DATA

4.2.3 Application

As an application example, *Customer* entities and their associated *order* entities can be retrieved efficiently in one, join-less, **select** operation (Script 4.4). The result of these queries is shown in Figure 4.2. At the physical level, this model allows indexes to be defined on columns that appear in different entity types, providing a function somewhat similar to Oracle *clusters*.

```

select  CustID, Name, Address, City, OrdID, DatOrd
from    DATA
where   EType in ('CUST', 'ORD')
and     CustID = 'C400';

select  CustID, Name, Address, City, OrdID, DatOrd
from    DATA C, DATA O
where   C.EType = 'CUST'
and     O.EType = 'ORD'
and     C.CustID = 'C400'
and     O.CustID = C.CustID;

```

Script 4.4 - Extracting data of customer C400 and his orders in join-less query (top) and its join-based equivalent

Ety	EType	CustID	Name	Address	City	Cat	Acc	OrdID	DateOrd	Qord	ProdID	Description	Price	QonH
1	CUST	B112	HANGENNE	23, r. Dumont	Poitiers	C1	1250							
2	CUST	C123	MERCIER	25, r. Lemaitre	Namur	C1	-2300							
3	CUST	B332	MONTI	112, r. Neuve	Genève	B2	0							
4	CUST	F010	TOUSSAINT	5, r. Godefroid	Poitiers	C1	0							
5	CUST	K111	VANDELST	180, r. Florimont	Lille	B1	720							
6	CUST	S127	VANDERKA	3, av. des Roses	Namur	C1	-4580							
7	CUST	B512	GILLET	14, r. de l'Été	Toulouse	B1	-9700							
8	CUST	C062	ROFFIN	22, r. de la gare	Namur	B2	3200							
9	CUST	C062	FERARD	45, r. de l'Église	Poitiers	B2	3200							
10	CUST	C003	AVRON	9, ch. de l'Église	Toulouse	B1	-1700							
11	CUST	K729	NEUMAN	40, r. Bransart	Toulouse	B1	0							
12	CUST	F011	PONCELET	17, Clôds des Erables	Toulouse	B2	0							
13	CUST	L422	FRANCK	60, r. de Wépion	Namur	C1	0							
14	CUST	S712	GUILLAUME	14a, ch. des Roses	Paris	B1	0							
15	CUST	D063	MERCIER	201, bvd du Nord	Toulouse		-2250							
16	CUST	F400	JACOB	78, ch. du Moulin	Bruxelles	C2	0							
17	ORD	K111	--	--	--	--	--	30178	2013-12-21	--	--	--	--	--
18	ORD	C400	--	--	--	--	--	30179	2013-12-22	--	--	--	--	--
19	ORD	S127	--	--	--	--	--	30182	2013-12-23	--	--	--	--	--
20	ORD	C400	--	--	--	--	--	30184	2013-12-23	--	--	--	--	--
21	ORD	F011	--	--	--	--	--	30185	2014-01-02	--	--	--	--	--
22	ORD	C400	--	--	--	--	--	30186	2014-01-02	--	--	--	--	--
23	ORD	B512	--	--	--	--	--	30188	2014-01-03	--	--	--	--	--
24	DET	--	--	--	--	--	--	30178	--	25	CS464		--	--
25	DET	--	--	--	--	--	--	30179	--	20	PA60		--	--
26	DET	--	--	--	--	--	--	30179	--	60	CS262		--	--
27	DET	--	--	--	--	--	--	30182	--	30	PA60		--	--
28	DET	--	--	--	--	--	--	30184	--	120	CS464		--	--
29	DET	--	--	--	--	--	--	30184	--	20	PA45		--	--
30	DET	--	--	--	--	--	--	30185	--	15	PA60		--	--
31	DET	--	--	--	--	--	--	30185	--	600	CS222		--	--
32	DET	--	--	--	--	--	--	30186	--	260	CS464		--	--
33	DET	--	--	--	--	--	--	30186	--	2	PA45		--	--
34	DET	--	--	--	--	--	--	30188	--	70	PA60		--	--
35	DET	--	--	--	--	--	--	30188	--	92	PH222		--	--
36	DET	--	--	--	--	--	--	30188	--	180	CS464		--	--
37	DET	--	--	--	--	--	--	30188	--	22	PA45		--	--
38	PROD	--	--	--	--	--	--	--	--	--	CS262	RAFT. PINE 200x6x2	75	45
39	PROD	--	--	--	--	--	--	--	--	--	CS264	RAFT. PINE 200x6x4	120	2690
40	PROD	--	--	--	--	--	--	--	--	--	CS464	RAFT. PINE 400x6x4	220	450
41	PROD	--	--	--	--	--	--	--	--	--	PA45	NAILS STEEL 45 (1K)	105	580
42	PROD	--	--	--	--	--	--	--	--	--	PA60	NAILS STEEL 60 (1K)	95	134
43	PROD	--	--	--	--	--	--	--	--	--	PH222	PL. BEECH 200x20x2	230	782
44	PROD	--	--	--	--	--	--	--	--	--	PS222	PL. PINE 200x20x2	185	1220

Figure 4.1 - Contents of the Universal table associated to the ORDERS database

CustID	Name	Address	City	OrdID	DateOrd
C400	FERARD	65, r. du Tertre	Poitiers	--	--
C400	--	--	--	30179	2013-12-22
C400	--	--	--	30184	2013-12-23
C400	--	--	--	30186	2014-01-02

CustID	Name	Address	City	OrdID	DateOrd
C400	FERARD	65, r. du Tertre	Poitiers	30179	2013-12-22
C400	FERARD	65, r. du Tertre	Poitiers	30184	2013-12-23
C400	FERARD	65, r. du Tertre	Poitiers	30186	2014-01-02

Figure 4.2 - Result of the execution of queries 4.4

4.2.4 Origin of the Universal table model

This data model is derived from the concept of *Universal relation* of the relational theory, though the two constructs are quite different. The *Universal table* is the union of source tables while the *Universal relation* would be obtained by outer joins of the source tables. One of the main goal of the *Universal relation* is to offer a simpler relational algebra, devoid of the join operator.

4.2.5 Comments

This model brings much flexibility for schema evolution. For example,

- Since all columns are nullable, new columns can be added without impact on the contents of the table,
- Each entity can be structured independently of the others of its type (e.g., a definite customer entity can be assigned a value of column DESCRIPTION),
- New entity types can be added without structural modification, provided the required columns are available.

Such loosely structured data model is appropriate when the nature and the structure of entities are rapidly evolving.

However, it does not provide an easy support for the native SQL integrity constraints (unique keys, foreign keys, not null column). First conclusion: more flexibility implies lesser expressivity of the schema. More on this later.

The scripts of this section are available in file **Schemaless1-Universal.sql** in directory **Case_Schemaless**.

4.3 Column-oriented data model

This model represents *each property by a specific table* and can be considered the opposite of the *Universal table* model. As compared to the four source tables of database ORDERS.db, this model will include one table (called *column-table*) per source column. For instance, the data of source table CUSTOMER are distributed among five column-tables, each comprising the primary key (**CustID**) plus one source column, respectively for Name, Address, City, Cat, Account values. Primary key CustID is common to these tables to allow the reconstruction of source table CUSTOMER through joins. The second column is simply named **Value**, so that all the column-tables (but one!) deriving from a source table have the same schema. Script 4.5 creates the tables for storing the data of database ORDERS.db.

```
create table CUST_NAME(
  CustID    char(10) not null primary key,
  Value     varchar(32) not null);
create table CUST_ADDRESS(
  CustID    char(10) not null primary key,
  Value     char(60) not null);
create table CUST_CITY(
  CustID    char(10) not null primary key,
  Value     char(30) not null);
...
create table ORD_CUSTID(
  OrdID     char(12) not null primary key,
  Value     char(10) not null);
create table ORD_DATEORD(
  OrdID     char(10) not null primary key,
  Value     date not null);
...
```

Script 4.5 - Creating column-tables

4.3.1 Generating the Column-oriented data from the source tables

Script 4.6 migrates the data from the source database. Figure 4.3 shows the contents of tables **CUST_NAME** and **CUST_ADDRESS**.

We observe that we have not created a specific table for the primary key of the source tables. Indeed, its values are stored in each of the column-table. This structure is valid provided (1) the source table comprises at least one column in addition to the primary key and (2) *null* values are also represented. If condition (1) is not satisfied, then the source table can be taken as a degenerated column-table.

```

insert into CUST_NAME(CustID,Value)
  select CustID,Name from CUSTOMER;
insert into CUST_ADDRESS(CustID,Value)
  select CustID,Address from CUSTOMER;
insert into CUST_CITY(CustID,Value)
  select CustID,City from CUSTOMER;
...
insert into ORD_CUSTID(OrdID,Value)
  select OrdID,CustID from CUSTORDER;
insert into ORD_DATEORD(OrdID,Value)
  select OrdID,DateOrd from CUSTORDER;
insert into DET_QORD(OrdID,ProdID,Value)
  select OrdID,ProdID,Qord from DETAIL;
insert into PROD_DESCR(ProdID,Value)
  select ProdID,Description from PRODUCT;
insert into PROD_PRICE(ProdID,Value)
  select ProdID,Price from PRODUCT;
...

```

Script 4.6 - Loading column-tables

CustID	Value	CustID	Value
B112	HANSENNE	B112	23, r. Dumont
C123	MERCIER	C123	25, r. Lemaître
B332	MONTI	B332	112, r. Neuve
F010	TOUSSAINT	F010	5, r. Godefroid
K111	VANBIST	K111	180, r. Florimont
S127	VANDERKA	S127	3, av. des Roses
B512	GILLET	B512	14, r. de l'Eté
B062	GOFFIN	B062	72, r. de la Gare
C400	FERARD	C400	65, r. du Tertre
...

Figure 4.3 - Contents of column-tables CUST_NAME and CUST_ADDRESS

We observe that source table DETAIL is transformed into a column-table the schema of which is different from that of the other column-tables. Indeed, this table has a composite primary key and only one column in addition to its primary key. Therefore, the source table and the unique column-table are exactly the same, bar some of their column names.

4.3.2 From the Column-oriented tables back to the source tables

Script 4.7 recovers source table CUSTOMER by joining the tables that represent its columns.


```

select C1.CustID,C1.Value as Name,C2.Value as Address,
       C3.Value as City,C4.Value as Cat,C5.Value as Account
from CUST_NAME C1,CUST_ADDRESS C2,
     CUST_CITY C3,CUST_CAT C4,CUST_ACCOUNT C5
where C2.CustID = C1.CustID
and   C3.CustID = C1.CustID
and   C4.CustID = C1.CustID
and   C5.CustID = C1.CustID;

```

Script 4.7 - Rebuilding source table CUSTOMER from column tables

4.3.3 Column-oriented model with abstract primary keys

In the column-tables, column CustID is intended to denote entities. This technique is fine for CUSTOMER table for two reasons:

- the customer Ids are short and meaningless
- the primary key of the table comprises one column only.

Performance problems may arise for large primary keys, which are duplicated in each column-table. In addition, the primary indexes will also be quite large and therefore will entail lookup time penalties.

Composite primary keys are more tricky to cope with. In such cases, column tables are made up of more than two columns: at least two for the primary key + one. So, there is no common table format, which makes the model no longer uniform.

Hence the idea to denote entities with abstract Ids, as we did in table DATA of the *Universal table* model (Script 4.1). This way, we can define a column-table for each source column, be it a component of a primary key or not, making the representation quite uniform (Script 4.8). We observe that table CUST_CUSTID has a spacial structure: Column Entity is declared *auto increment* and a unique predicate is stated on column Value.

To load the column-tables, we proceed in two steps. First, we load the table derived from the primary key in order to generate the abstract Ids:

```

insert into CUST_CUSTID(Value)
select CUSTID from CUSTOMER order by CUSTID;

```

Then, we load the other column-tables with the abstract Ids that have been produced:

```

insert into CUST_NAME(Entity,Value)
select Entity,Name
from CUST_CUSTID I, CUSTOMER C where I.Value = C.CustID;

```

```

create table CUST_CUSTID(
    Entity    integer not null primary key autoincrement,
    Value     char(10) not null unique);
create table CUST_NAME(
    Entity    integer not null primary key,
    Value     char(32) not null);
create table CUST_ADDRESS(
    Entity    integer not null primary key,
    Value     char(60) not null);
create table CUST_CITY(
    CustID    integer not null primary key,
    Value     char(30) not null);
...
create table DET_PK(
    Entity    integer not null primary key autoincrement,
    Value1    char(12),
    Value2    char(15));
create table DET_ORDID(
    Entity    integer not null,
    Value     char(12) not null);
create table DET_PRODID(
    Entity    integer not null,
    Value     char(15) not null);
create table DET_QORD(
    Entity    integer not null,
    Value     integer not null);

```

Script 4.8 - Creating column-tables based on abstract primary keys

The case of composite primary keys will be tackled in a slightly different way. We create a special dictionary table the role of which is to generate the abstract Ids and to map the primary key composite values to these abstract values. For table `DETAIL`, it is declared as follows:

```

create temp table DET_PK(
    Entity    integer not null primary key autoincrement,
    Value1    char(12),
    Value2    char(15));

```

The dictionary table is filled by the primary key values of `DETAIL`:

```

insert into DET_PK(Value1,Value2)
select OrdID,ProdID from DETAIL order by OrdID,ProdID;

```

The column-tables are then loaded with the abstract Ids that have been produced:

```

insert into DET_ORDID(Entity,Value)
select Entity,OrdID from DET_PK K, DETAIL D
where K.Value1 = D.OrdID and K.Value2 = D.ProdID;

```

The uniqueness property of the source primary key of table CUSTOMER is ensured by the **unique** predicate in column-table CUST_CUSTID.

Unfortunately, the expression of composite primary keys is no longer straightforward, since they hold on joins of column tables. The following query lists the primary key values that are not unique among the column-tables of DETAIL:

```
select O.Value as OrdID,P.Value as ProdID,count(*) as Duplicate
from   DET_ORDID O,DET_PRODID P
where  O.Value = P.Value
group by O.Value,P.Value
having Duplicate > 1;
```

4.3.4 About performance

We have measured the time needed to load the column-tables and to rebuild the source table for a CUSTOMER table containing 32,768 rows (= **32K** rows). These figures are collected in Figure 4.4 (times in **ms.**). If the first technique, based on source primary keys, is a bit faster for the generation of column-tables, the second technique, based on abstract Ids, appears to be *four times faster* for rebuilding the source table rows. These times are linear *wrt* the table size: doubling the number of rows doubles the execution times.

Entity Id	Load time	Rebuild time
primary key	279	167
abstract entity id	318	42

Figure 4.4 - Execution time of the main conversion processes for the two approaches

4.3.5 Not all Column-oriented models are created equal

The term *column-oriented* also names a family of so-called NoSQL data managers, the main representative of which certainly is *Cassandra*. In the latter case, this name is misleading. The original data model of *Cassandra* is particularly convoluted, using seemingly new concepts under proprietary vocabulary. Actually, this historical model is being abandoned and replaced by an alternative model, CQL, made up of concept like keyspace (= database), type, table, column, primary key, index, trigger as well as operators select, insert, update and delete, that is, ironically, ... a subset of SQL!

4.3.6 Comments

This model may bring several advantages. First, the evaluation of a query only gets access to the columns that are relevant and to no other ones. This reduces the volume of data read from the external memory. This is particularly important for tables that comprise many, possibly large, columns. Another advantage is that, by storing the columns on different devices, their access can be performed in parallel, thus reducing response time.

On the other hand, the expression of some common integrity constraints may prove complex.

The way *null* values are represented is worth a short discussion. The first technique, that has been chosen in this discussion, consists in inserting, for each source row, a row in each column-table, be its value *null* or not *null*. According to another way, *null* values are represented by a missing row in the column table. Both approaches are consistent but when we rebuild the source table, or a part of it, the first representation will need *inner joins* while the second one will require *outer joins*.

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

A discussion and links to *column-oriented database managers* can be found in reference: http://en.wikipedia.org/wiki/Column-oriented_DBMS.