Chapter 16
Object Oriented Databases

We have so far discussed quite a few things about relational databases. Indeed, RDB has been widely used in modeling and manipulating data since 1980’s because of the simplicity of the well-defined relational data model and its appropriateness. Tables are just the right representation for much of the data in the business world.

On the other hand, the relational data model is not as appropriate for some of the other “non-traditional” applications such as CAD, GIS etc..

Even for the business oriented data, RDB is not perfect.
A sticking point

Assume that, when working with the design stage, we come up with an entity $E(A, B, C)$, where $A$ is a key. Thus, by definition, we immediately have an FD: $A \rightarrow B, C$.

If none of $B$ and $C$ is set value, we can easily convert it into an RDB table. Otherwise, assume $B$ is set valued, since every RDB table is in 1NF, $A$, $B$ have to become a superkey. Then, since the left-hand side of the above FD is a strict subset of a superkey, $E(A, B, C)$ is not even in 2NF.

We then have to normalize the above relation before anything can be done.

**Question:** Is 1NF really a sacred cow?

**Answer:** It might not be for some application.
An example

Consider the following relational schema that describes people:

Person(SSN:String, Name:String, PhoneN:String, Child:String)

Since a person could have several phone numbers and several children, the key of this relation has to be a combination of SSN, PhoneN, and Child. A similar situation occurs with email addresses.

Below is an instance of this schema:

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>PhoneN</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-123-4567</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-345-6789</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-123-4567</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-345-6789</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-6543</td>
<td>444-55-6666</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-1111</td>
<td>555-66-7777</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-6543</td>
<td>555-66-7777</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-1111</td>
<td>444-55-6666</td>
</tr>
</tbody>
</table>
You know what’s wrong, don’t you?

This table is not in 3NF, thus not in BCNF, since one of its FDs: \( SSN \rightarrow \text{Name} \), violates the condition: \( SSN \) alone is not a key of this table; and \( \text{Name} \) is not part of the key.

We can certainly go through the normalization process to decompose this schema to three sub-schemas:

\[ \text{Person}(SSN, \text{Name}), \text{Phone}(SSN, \text{PhoneN}), \text{and} \text{ChildOf}(SSN, \text{Child}) \]

This leads to the following instances:

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Joe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>PhoneNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>516-123-4567</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>516-345-6789</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>212-987-6543</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>212-987-1111</td>
</tr>
</tbody>
</table>
Although all the redundancy have been taken out, this design is tough to work with. 😞

Consider the following query “get the phone numbers of Joe’s grandchildren.,” the following gets it from the original table,

Select G.PhoneN From Person P, Person C, Person G
Where P.Name=’Joe Public’ And P.Child=C.SSN
And C.Child=G.SSN

while the following gets it from the decomposition

Select N.PhoneN From ChildOf C, ChildOf G, Person P, Phone N
Where P.Name=’Joe Public’ And P.Child=C.SSN And
C.Child=G.SSN And G.SSN=N.SSN

Both are awkward: It meets user’s needs, but not user friendly.
Where are the tissues?

One issue is that both phone number and children are fundamentally set valued, and the RDB is not good at handling this sort of data: Anything that can handle set-valued data is not even 1NF. 🙁

A much more appropriate schema for the original table might be the following:

\[
\text{Person} \left( \text{SSN:} \text{String}, \text{Name:} \text{String}, \right. \\
\left. \text{PhoneN:} \{\text{String}\}, \text{Child:} \{\text{String}\} \right),
\]

where “\{\}” indicates set-valued attributes. Thus, e.g., the type for Child tells that the value for this attribute is a set. One of its tuples looks like the following:

\[
(111-22-3333, \text{Joe Public}, \\
\{516-123-4567, 516-345-6789\}, \\
\{222-33-4444, 333-44-5555\})
\]
Another one is...

that it is awkward to query such tables.

Assume that the type of Child is Person rather than String, then an SQL query can treat the value of the Child attribute as a set of Person.

For example, the expression P.Child.Child means children of children of a person P, thus her grand children.

Thus, it will be possible for us to formulate a much more natural query as follows:

Select P.Child.Child.PhoneN
From Person P
Where P.Name='Joe Public'

**Question:** Why didn’t I put in under “mysql>”? 
Yet another...

Assume that some, but not all, people in our Person database are students. Since a student is a person, we can represent this fact by drawing an arrow from the Student entity to the Person entity as we did when working with an E/R diagram, back when?

Since RDB does not support the IsA concept, we have to do a simulation by throwing out the general information, such as Name, to the Person entity and keep only those specific for students in the Student entity.

This leads to the following structure for the Student relation.

Student(SSN:String,Major:String)
What to say?

**Question:** Will the following query work, if we want to get the names of all the CS majors?

```
Select S.Name From Student S
Where S.Major='CS'
```

**Answer:** I would expect you to say “No”, although you might think otherwise, since every student is a person, she should have a name.

**Question:** What should we say?

**Answer:** We have to get someone’s name from the `Person` table via an expensive natural join:

```
Select P.Name From Student S, Person P
Where S.Major='CS' And P.SSN=S.SSN
```

**Question:** Is the order of the conjuncts in the `Where` clause important?

**Answer:** Depends on DBMS.
A few examples

We will look at a few examples in the context of SQL:1999 and SQL:2003.

Crate Table Person (
   Name Char(20),
   Address Row(Number Integer,Street Char(20),Zip(Char(5)))
)

We can then refer to a value in a tuple using the path expression, e.g., as follows:

Select P.Name From Person P
Where P.Address.Zip='11794'

Insert Into Person(Name,Address)
Values(‘John Doe’,Row(666,’Russell St.’,’03264’))

Update Person
Set Address=Row(21,’Main St.’,’03264’)
Where Address=Row(666,’Russell St.’,’03264’)
   And Name=’John Doe’
Let's have our own

By SQL:1999/2003, we can define our user defined type (UDT). We can only define variables and procedures with MySQL 5.7.

For example,

Create Type PersonType As (  
    Name Char(20),  
    Address Row(Number Integer, Street Char(20), Zip(Char(5)));

Create Type StudentType Under PersonType As (  
    Id Integer,  
    Status Char(2))  
Method award_degree() Returns Boolean;  
Create Method award_degree() For StudentType  
Language C  
External Name 'file:/home/admin/award_degree';

Here, we define StudentType as a subtype of PersonType, with the Under clause. It then inherits all the properties of PersonType.

In addition, StudentType has its own type, plus a signature for a method, which is then defined using the Create Method...For... clause, telling where its implementation can be found.
Now the tables

We can do business as usual, e.g.

Create table Transcript (  
    Student StudentType,  
    CrsCode Char(6),  
    Semester Char(5),  
    Grade Char(1))

We can also create a table using a UDT, where the table must have the exact structure as given in the UDT:

Create Table Student of StudentType;

The only way to create an object in SQL is via an insertion into a table, whence it is given its unique object id, and the table itself will be regarded as a class, and the set of the objects as its extent.
Inheritance

Since StudentType is a subtype of PersonType, although such attributes of Name, Address are not explicitly mentioned in the StudentType, they are implicitly inherited.

Thus, if we create the Student table as follows:

```sql
Create Table Student Of StudentType Under Person;
```

then, when we insert a tuple into the Student table, as follows:

```sql
Insert Into Student(Name,Address,Id,Status)
Values
(‘John Jones’,Row(123,’Main St.’,11733),1122,’Junior’)
```

it will be automatically inserted into the Person table as well.

Notice that in this case, the Person table has become a super table of the Student table.
OOP in PhP

PhP supports object-oriented programming. We just look at a few simple examples. For details, please check out Chapter 20 of the Bible.

The following defines a class and then creates an instance.

class TextBox {
    var $body_text="my text";
    //the constructor
    function _constructor($text_in){
        $this->body_text=$text_in;
    }
    function display(){
        print("<table border=1><tr><td>
            $this->body_text
        </td></tr></table>" );
        print("</td></tr></table> ");
    }
    //Let’s create an instance.
    $box=new TextBox("customer box");
    $box->display();"
An inheritance example

We now declare a child object, extending the just defined TextBox:

class TextBoxHeader extends TextBox{
    var $header_text;
    //constructor
    function _constructor($head_text_in, $box_text_in){
        $this->header_text=$head_text_in;
        $this->body_text=$body_text_in;
    }
    function display(){
        //its code
    }
    function make_header($text){
        return($text);
    }
    function make_body($text){
        return($text);
    }
}

NoSQL

As we have been seeing throughout this course, RDB consists of well-structured tables.

With recent development, particularly, in the area of big data, we want to have more flexible representation of data. NoSQL, meaning “non SQL”, “non relational”, or “not only SQL”, database provides a mechanism to store and retrieve data, not modeled in the tabular forms. Instead, it makes use of such data structures as key-value (associative array), wide column, graph, and document.

Depending on applications, NoSQL databases compromise consistency in exchange of availability, partition tolerance, and speed.

For more details, check out the relevant links on the course page, as well as the upcoming new course, CS 3650 Big Data Analytics, to start in Fall 2018. 😊