THE 19TH EDITION OF THE INTERNATIONAL CONFERENCE
EUROPEAN INTEGRATION
REALITIES AND PERSPECTIVES

Relational-Object Mapping of Databases

Radu Bucea Manea Tonis[1]

Abstract: Mapping relational databases is a programming technique that enables developers to interact with the database using objects, which are instances of classes defined in a specific programming language. **Objectives**: This paper propose a scientific method based on several rules in order to transform the logic layer of a relational database into an object model formalized in a class diagram. **Prior Work**: This research is based on C.J. Date logic and relational approach on modelling relational databases. **Approach**: We are using the high-level Microsoft C# object-oriented programming language in order to implement the object model of a classic transactional database employing a set of rules and a specific library for serializing collections of objects for a dedicated (MongoDB) persistence layer, **Results:** the JSON serialized form of the objects is further used to generate new instances according to the business application layer. **Implications**: The back & forward transformations without information loss by serializing and deserializing objects according to a predefined data scheme validates the methodology proposed by this paper. **Value**: The importance of our research resides in the plenitude of source code utilized for exemplifying our methodology and the diversity of real-life situations that might benefit of our research in terms of efficiency and consistent data management.

Keywords: normalization; functional dependency; CAP theorem.

JEL Classification: technological change; the knowledge-based economy (O3).

1. Introduction

That part of logic that has to do with the study of sentence forms (with logical forms) is called “philosophical logic” by Russell. The author emphasizes the importance and need to study forms: “... a certain kind of knowledge of logical forms, even if in most people it is not explicit, is involved in the whole understanding of speech.” (Russell, 2009)

In the case of the form of representation, including the forms of elementary sentences, Wittgenstein identifies the propositional form with the variable, according to “the general propositional form is a variable” from Tractatus. (Wittgenstein, 2001)

Alonzo Church draws attention to the choice of a system of logical notation and analysis over substitution of special symbols for terms or sentences of natural language, as follows: “to adopt a certain language formalized implies (...) the adoption of a particular theory or system of logical analysis.” (Church, 1996)

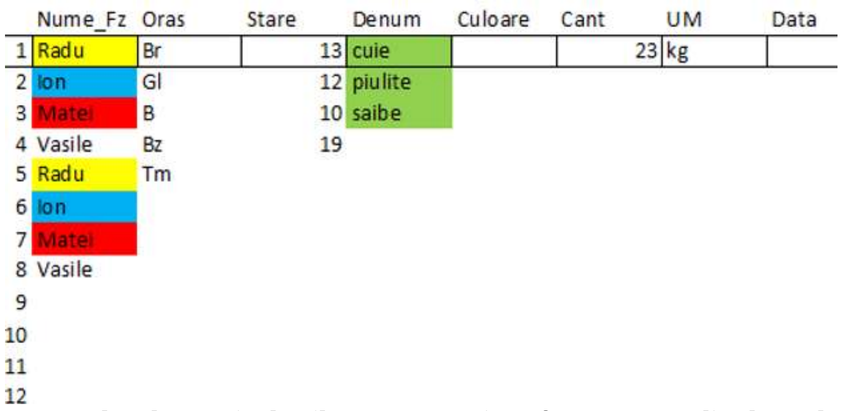
[1]Senior Lecturer, Danubius International University, Romania, Address: 3 Blvd Galati, Galati 800654, Corresponding author: radumanea@univ-danubius.ro.

Converting logical forms to normalized databases is a fundamental process in database design.

2. Database Design

2.1. Database Creation

We determine first the entities (objects or concepts) in the logical form and the relationships between them. Further on, each entity becomes a table in the database schema, attributes of entities become columns in the respective tables and we represent relationships between entities using pairs made of primary (PK) and foreign keys (FK).



	Nume_Fz	Oras	Stare	Denum	Culoare	Cant	UM	Data
1	Radu	Br	13	cuie		23	kg	
2	Ion	Gl	12	piulite				
3	Matei	B	10	saibe				
4	Vasile	Bz	19					
5	Radu	Tm						
6	Ion							
7	Matei							
8	Vasile							
9								
10								
11								
12								

Figure 1. The Flat or Single File Representation of a Pre-Normalized Database

This logical form of a database is sub-optimal in terms of storage efficiency – data must be redundantly stored – and it lacks any indexes for sorting, filtering, etc. Thus, we must apply the normalization process.

2.2. Database Normalization

Normalization is the process of organizing the attributes and tables of a relational database to minimize redundancy and dependency. First Normal Form (1NF) eliminates repeating groups by putting each attribute in a separate column so each column has atomic (indivisible) values. We observe from Figure 1 that our flat structure is designed accordingly to 1NF.

Second Normal Form (2NF) removes partial dependencies by creating separate tables for sets of values that apply to multiple records. We may accomplish this simply by identifying functional dependences as might be seen in Figure 2:

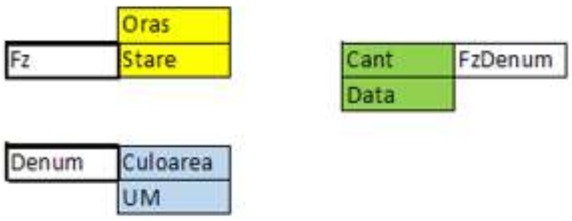


Figure 2. Functional Dependency Applied to Our Flat Initial Structure

Third Normal Form (3NF) eliminates transitive dependencies by removing attributes not dependent on the primary key, accordingly to Figure 3:

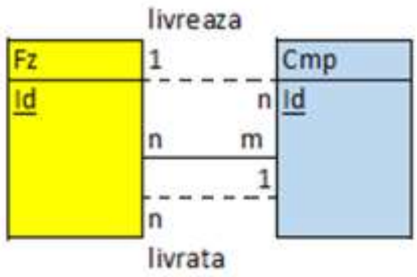


Figure 3. Multi-value Dependency (Many-to-Many) between Furnizori (Fz) and Componente (Cmp) Tables

Further normalization eliminates anomalies related to functional dependencies (BCNF) and address multi-valued dependencies or other advanced issues (4NF), as may be seen in Figure 4:

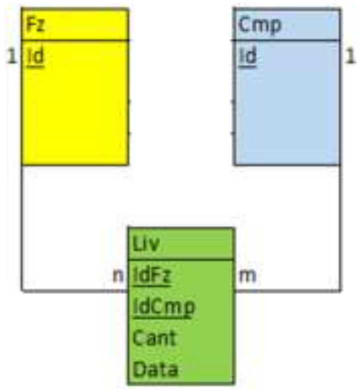


Figure 4. Normalized Transactional (Livrari) Database

Basically, the multi-valued dependency was resolved/mapped into a new relation variable (Liv) consisting on the foreign keys (FKs) as multi-attribute primary key (IdFz & IdCmp) and several common attributes (Cant, Data).

In conclusion, normalization ensures efficient storage, minimizes redundancy, and maintains data integrity in relational databases. Thus, it's a crucial step in creating robust and scalable database systems.

2.3. Relational – Object Mapping (ROM)

The normalized schema stands for efficiency, data integrity, and adherence to business rules. Further on, we have to adjust the design as needed based on performance considerations, user feedback, and evolving requirements. For this purpose, we have to apply the following transformation rules:

- A tabular structure becomes a class;
- A field becomes an attribute/property;
- A record becomes an instance;
- A table becomes a collection;
- An association can be transformed into:
  - Simple pairing;
  - Inheritance/specialization ratio
  - Aggregation/composition relationship

We have successfully applied the rules on the normalized database schema obtaining the UML diagram in the Figure 5:

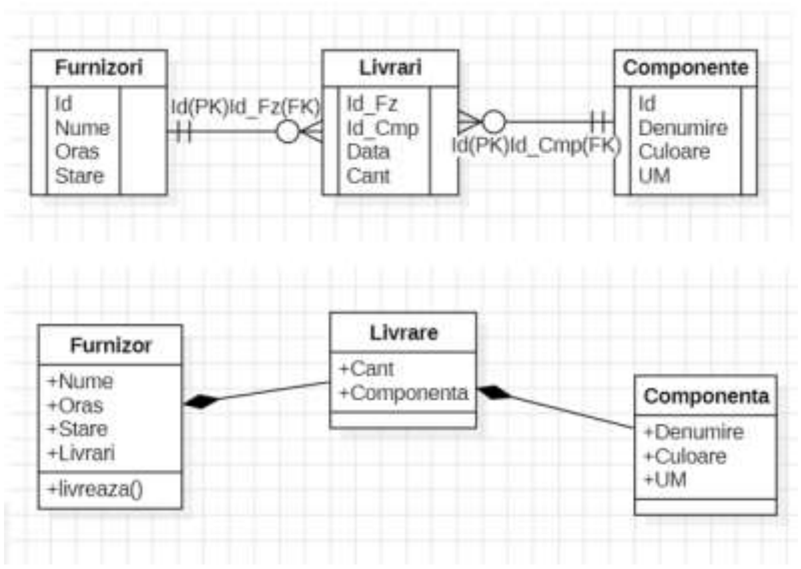


Figure 5. The ERD and UML Diagrams According to Transformation (ROM) Rules

Overall, ROM simplifies database interactions by abstracting away the complexities of SQL and providing a more intuitive way to work with databases in object-oriented programming languages. However, it's important to understand the underlying SQL and database concepts to use it effectively, avoid common pitfalls and convert data consistently between incompatible type systems.

3. Persistence Layer of the Objectual Data Model

3.1. Mapping Tables into Classes

For mapping tables into classes, we have used the 2022 version of Microsoft Visual Studio Community development platform. Additionally, we have installed individually the Class Designer to show the UML style class diagram, please see Figure 6 accordingly:

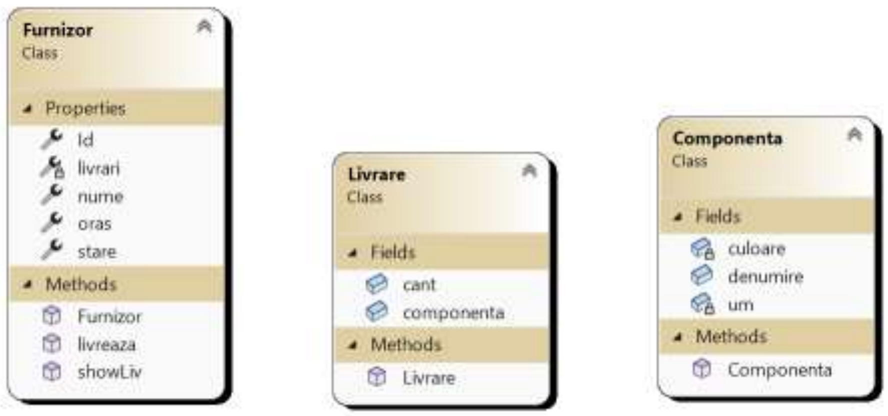


Figure 6. UML Style Class Diagram Auto-Generated by Visual Studio’s Dedicated Component

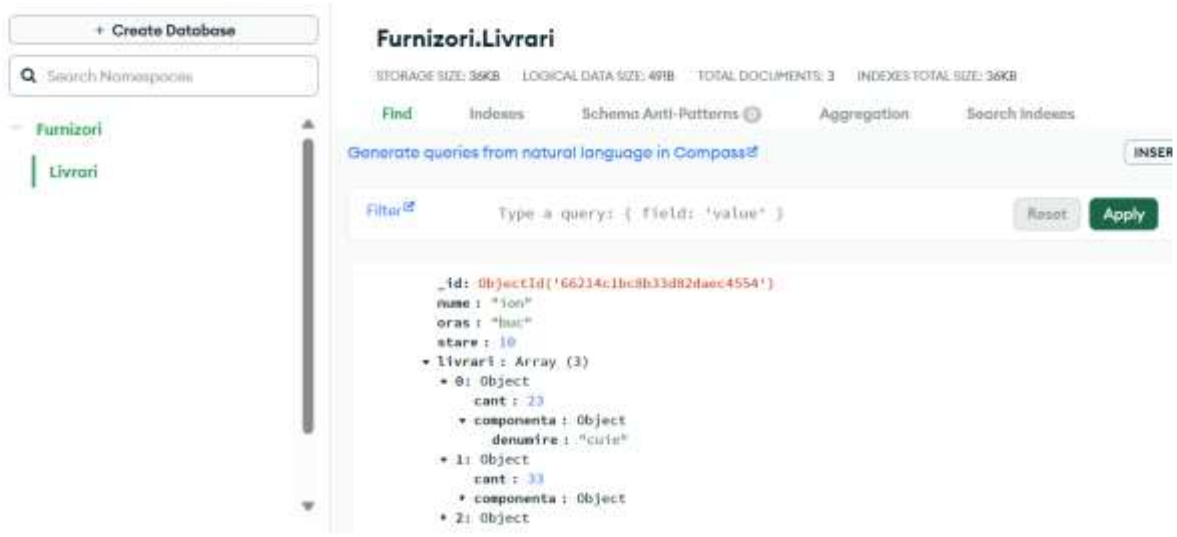
3.2. Serializing Instances (Objects) into JSON Format File

Further on, we have opted for Newtonsoft library in order to serialize instances obtained from the relational database server (MySQL):

```
// open a connection asynchronously
using var connection = new MySqlConnection(builder.ConnectionString);
await connection.OpenAsync();
// create a DB command and set the SQL statement without parameters
using var command = connection.CreateCommand();
command.CommandText = @”SELECT * FROM Furnizori;”;
// execute the command and read the results
using var reader = await command.ExecuteReaderAsync();
while (reader.Read())
{
   var name = reader.GetString(“numef”);
   var oras = reader.GetString(“oras”);
   var stare = reader.GetInt32(“stare”);
   Furnizor fz_db = new Furnizor(name, oras, stare);
}
Console.WriteLine(JsonConvert.SerializeObject(fz));
fz.showLiv();
```

The output is presented bellow in the form of a JSON file ready to be uploaded as a single document into MongoDB server, please fallow Figure 7 to see the result below:

```
{
   “nume”: “ion”,
   “oras”: “buc”,
   “stare”: 10,
   “livrari”: [
      {
         “cant”: 23,
         “componenta”: {
            “denumire”: “cuie”
         }
      },
      {
        “cant”: 33,
        “componenta”: {
          “denumire”: “suruburi”
        }
      },
      {
        “cant”: 43,
        “componenta”: {
          “denumire”: “piulite”
        }
      }
    ]
}
```


+ Create Database
Search Namespaces
Furnizori
Livrari
Furnizori.Livrari
STORAGE SIZE: 36KB LOGICAL DATA SIZE: 491B TOTAL DOCUMENTS: 3 INDEXES TOTAL SIZE: 36KB
Find
Indexes
Schema Anti-Patterns
Aggregation
Search Indexes
Generate queries from natural language in Compass
INSER
Filter
Type a query: { field: 'value' }
Reset
Apply

```
   _id: ObjectId('66214c1bc8b33d82daec4554')
   nume : "ion"
   oras : "buc"
   stare : 10
 ▾ livrari : Array (3)
   ▾ 0: Object
       cant : 23
     ▾ componenta : Object
         denumire : "cuie"
   ▸ 1: Object
       cant : 33
     ▸ componenta : Object
   ▸ 2: Object
```

Figure 7. MongoDB Document in the Livrari Collection from the Furnizori Database

3.3. Repatriating Data by Deserializing MongoDB Documents

The server manages documents with auto-generated unique id’s so there’s two strategies in order to parse correctly the resulted JSON file. The first one uses REGEX (or other programming means) to remove the field (_id: ObjectId) inserted by the server into the document, the second must update the related class definition with one more new property (e.g. public string Id { get; set; }) annotated with [BsonId]/[BsonRepresentation(BsonType.ObjectId)].

For data repatriation in object form we’ve opted for the second strategy, thus choosing the BSON default tool provided by the MongoDB 2.24.0 driver version:

var connectionString = Environment.GetEnvironmentVariable(“MONGODB_URI”);

```
var client = new MongoClient(connectionString);

var collection = client.GetDatabase(“Furnizori”).GetCollection<BsonDocument>(“Livrari”);

var filter = Builders<BsonDocument>.Filter.Eq(“nume”, “ion”);

var document = collection.Find(filter).First().ToString();

Furnizor fz = BsonSerializer.Deserialize<Furnizor>(document);
```

4. Conclusion

Relational-object mapping is not perfect and often results in loss of information. It is good to be noted that persistence pattern is hierarchical (consistent), distributed (portable) and no longer suitable for updates (not accessible anymore), according to CAP Theorem regarding NoSQL databases.

References

Church, A. (1996). *Introduction to Mathematical Logic*. Princeton: Princeton University Press.

Russell, B. (2009). *Our Knowledge of the External World*. London: Taylor & Francis.

Wittgenstein, L. (2001). *Tractatus Logico-Philosophicus*. Boca Raton: Routledge.