Object-oriented databases and db4o

ULB: Advanced Databases

David Gràcia - Sergi Nadal

Contents

1 Why object-oriented databases?

This first chapter will be an introduction on the origin and motivation of object-oriented databases, we will see which events triggered its development.
Besides that, we will also see its point of strength and its weaknesses compared to object-relational databases.

1.1 A look back

By the end of 1960, Edgar F. Codd presented "*A Relational Model of Data for Large Shared Data Banks*", in this paper he proposed a new database model. This new model was aimed to solve problems that appeared with the evolution of technology, applications and the amount of data they were dealing with. New innovative concepts were proposed like:

- Concept of table with structured data by type (columns) and tuples or registers (rows).
- Table relationships as primary key - foreign key.
- Formal foundation on set theory with algebraic and calculus-based techniques for querying.

With all the new possibilities relational databases offered, applications with new domains, demands and requirements appeared.

On the other hand, in parallel object oriented programming languages (OOPL) began to develop with new functionalities like: user-defined classes, methods, encapsulation, ... Those OOPL have the ability to map very complex conceptual models into classes and relationships between them, the main problem comes when all this objects in memory have to be made persistent in the relational model, also known as impedance mismatch.

1.2 Impedance mismatch

The term impedance mismatch refers to the set of problems that rise from the combination of SQL and a host language.
Some of it's main characteristics are the following:

1.2.1 Mix of declarative and procedural programming languages

While most industry applications are programmed in a procedural language (OOPL is an example), SQL is a declarative language.
These two different programming paradigms do not fit together, hence extra tools are needed in order to work with them together.

In addition, interacting with SQL from Java is not straightforward neither from the developer's point of view (approximately 30% of the code and effort is used in conversion [5]) nor from the machine's (SQL cannot be interpreted by the compiler).

1.2.2 Different data models

Even the very same model can be represented in totally different ways in both objects and tables.
For example, a clear case it's a many-to-many relationship. Whereas in the relational model an intermediate table is needed to represent it, there doesn't exist such an intermediate class in object oriented languages.

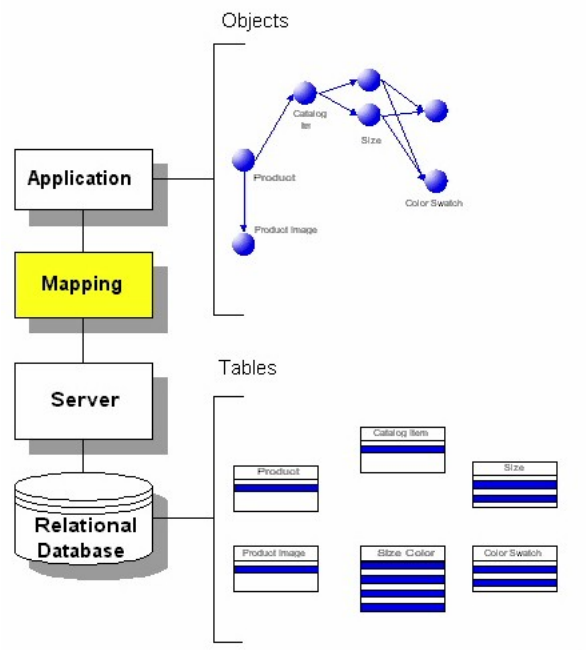

Figure 1.1: Mapping from classes to relations, the same model in a very different way. Source [7]

1.2.3 Different set of data types

Either on primitive types (integer, double, ...) nor on complex types (set, map, bag, ...) there's not a direct mapping. In the first case it may be necessary to use a different primitive because OOPLs are richer in primitives than SQL. However, in the case of complex types it may be necessary to change the physical model to suit this complex types in SQL in order to respect the first normal form.

1.3 ORDB: Object-Relational Database Systems

Seeing all the problems and inconvenients that integrating OOPLs and SQL lead to, a new group of databases was created in order to improve this integration between objects and SQL, those were known as Object-Relational Databases (ORDB) and were intro-

duced on the Object-Relational manifesto [2].

In this manifesto three main tenets and thirteen propositions were proposed that ORDBs should satisfy. The three tenets are:

- **Provide support for rich object structures and rules**

  Rich object structures are complex data elements such as arrays, sets and maps but also refer to text data and spatial data.

  On the other side, developer should be able to create a set of rules on data elements, records and collections, e.g. constraints for referential integrity.

- **Must subsume second generation DBMSs**

  The term second generation DBMSs refers to classic relational databases. This tenet refers to the fact that second generation DBMSs made a major contribution in two main aspects: non-procedural access and data independence, hence those should be maintained in ORDBs.

- **Must be open to other subsystems**

  Prospect to new functionalities: like decision support tools, access from many programming languages, interfaces to business graphic packages, ability to run application from different machine from the database and distribution of databases.

Next to those three tenets, the manifesto sets thirteen propositions which can fit in one of the three tenets. Those are as follows in figure 1.2.

The conclusion of this manifesto was that a new kind of database systems was required in order to fit the needs and integration of OOPLs, but keeping the same base as it was in relational model. In other words, there was a need of new tools to "hide" all the complexities this integration lead to.

<table>
<tr><th>Object and Rule Management</th><th>DBMS Function</th><th>Towards an Open System</th></tr>
<tr><td>Rich Type System</td><td>Non-procedural, high-level access languange</td><td>Accessible from multiple high-level languages</td></tr>
<tr><td>Multiple Inheritance</td><td rowspan="2">Suppport collections: enumeration of members or using the query language</td><td>For better or worse: SQL</td></tr>
<tr><td>Encapsulation</td><td>Enhancement of DBMS – programming language interfaces</td></tr>
<tr><td>UIDs and PKs</td><td>Updatable views</td><td rowspan="2">Queries and answers as the lowest level of communication (client / server)</td></tr>
<tr><td>Rules Enforcement</td><td>Data independence</td></tr>
</table>

Figure 1.2: Propositions for an ORDB, source [4].

1.4 OODB: Object-Oriented Database Systems

At the same time the object-relational manifesto was written, a different group of researchers with the same goal but a different point of view wrote the *Object-Oriented Database System Manifesto* [1]. In this paper they proposed some of the same propositions for this new type of database systems.

This manifesto was divided in characteristics of three different groups:

- **Mandatory**: those the system had to satisfy in order to be termed an object-oriented database system.
- **Optional**: the ones that could be added to make the system better, but were not mandatory.
- **Open**: those that the designer could make a number of choices.

The Golden Rules		Optional	Open
OO Data Model	DBMS		
Complex Objects	Persistence	Multiple Inheritance	Programming Paradigm
Object Identity	Secondary Storage Management	Type Checking and Inferencing	Representation System
Encapsulation	Concurrency	Distribution	Type System
Types / Classes	Recovery	Design Transactions	Uniformity
Inheritance	Ad hoc Query Facility	Versions	
Overriding, Overloading, Late Binding			
Extensibility			
Computational Completeness			

Figure 1.3: Propositions for an OODB, source [4].

Next list specifies the points in common and differences between O-R manifesto and O-O manifesto:

- **Common**
  - Encapsulation
  - DBMS main features: persistence, concurrency, recovery, ...
  - Open systems
- **Difference**
  - Object identity against object identifiers and primary keys.
  - High level programming language against SQL.
- **Not comparable**
  - Complex objects, types, extensibility or rich type system.
  - Inheritance or multiple inheritance.

As a conclusion, we can see that both ORDB and OODB try to achieve the same goal but in different ways, then why ORDBs have succeeded over OODBs?

The main reason is probably because major DB distributers already had a fully functional relational database (second generation), and adding new object support over it was easier and more secure than building a new database from scratch.
On the other hand, OODB manifesto it's a set of propositions but once companies started building these databases they found there was a lack of a common data model. There was, also, a lack of a strong theoretical framework and a lot of experimental activity was required. With the arrival of NOSQL OODBs are back, some examples are: *db4o*, *Objective/DB*, *ObjectStore*, *Versant's ODBMS*.

2 db4objects (db4o)

In this second chapter, we will see general aspects of db4o on very simple examples. In the second part we will use the same set of queries shown in LINQ at ADB course, and translate those to db4o in order to compare them.

2.1 Generals

In db4o, objects are stored directly as seen by the program, this means that the code (classes) is the database schema, there's no need for mappings.
This means that elements like object identity, associations, inheritance and complex objects are automatically stored without need of further configuration, this is known as orthogonal or transparent persistence.

In order to retrieve objects, there are two ways to do it:

- **By object identifier (OID)**: each object has associated an unique OID, same as RID on relational databases, and it is possible to directly retrieve an object by its OID.
- **By query**: it is possible to obtain objects querying any attributes of its class, there are three different ways of doing a query:
  - Query By Example (QBE)
  - Native Queries (NQ)

- SODA Query API

Other interesting features of db4o are as follows:

- **Deep prefetching**: this is the ability to configure the depth of the relation graph from an object to retrieve. In other words, how many navigation steps in relations will db4o perform and retrieve once an object is accessed.
- **Referential integrity**: automatically maintained by the system, no need to specify any constraint.
- **Inverse relationships**: automatically generated and updated when objects are added or removed.
- **Deletion**: objects are removed from all relations when deleted.

2.2 A simple use case

To view the basic functions of db4o, we will get a very simple model with two classes and do the basics in Java. The conceptual model is as follows in figure 2.1.

Figure 2.1: First simple model

```
1 public class Car {
2    public String brand;
3    public String model;
4 }
5
```

```
6  public class Person {
7     public String dni;
8     public String name;
9     public Car car;
10 }
```

The first thing to take into account is that db4o is not a DBMS one installs into a server, it comes as a *.NET* or *Java* library to attach into a project. Once it is correctly located, we initialize and open the database as follows (see comments in code):

```
1  import com.db4o.Db4oEmbedded;
2  import com.db4o.ObjectContainer;
3  import com.db4o.ObjectSet;
4  import com.db4o.config.EmbeddedConfiguration;
5
6  public static void main(String[] args) {
7     // Instantiate db4o library
8     EmbeddedConfiguration conf = Db4oEmbedded.newConfiguration();
9     // Activation and update depth for deep prefetching
10    conf.common().activationDepth(1);
11    conf.common().updateDepth(1)
12
13    // Open database testDB, if it does not exist a new one is created
14    ObjectContainer db = Db4oEmbedded.openFile(conf, "./testDB");
15 }
```

Storing an object is as simple as calling `store` or `update` operations.

```
1  Person p = new Person();
2  p.dni = "1234"; p.name = "Freddie Mercury"; p.car = null;
3  db.store(p);
4
5  Car c = new Car();
```

```
6  c.brand = "Volkswagen"; c.model = "Golf";
7  db.store(c);
8
9  p.car = c;
10 db.update(p); // New car automatically associated to person p
11
12 db.delete(p); // Also deletes car c depending on the configuration
```

As said, in order to query this data there are three methods available:

2.2.1 Query By Example

When using QBE a template object is provided to db4o and it will return which database objects match all non-default field values. This is done via reflection and combining all non-default values with `AND` expressions.

```
1 // Get Freedie back
2 Person p = new Person();
3 p.name = "Freddie Mercury";
4 ObjectSet result = db.queryByExample(p);
```

QBE has some important limitations such as:

- Inability to perform complex queries (`OR`, `NOT`, ...).
- Inability to query on values like 0, empty strings or `null`.
- Inability to use range conditions.
- Need to be able to create objects without initialized fields and a constructor for that.

2.2.2 Native Queries

This is the main db4o query interface, the main idea is to program a method that will be called with every instance of a certain class as parameter. These expressions should return true when an specific instance is to belong to the result set.

```
List <Person> persons = db.query(new Predicate<Person>() {
   public boolean match(Person p) {
      return p.name.equals("Freddie Mercury");
   }
});
```

2.2.3 SODA Query API

This API allows direct access to nodes of query graphs. A query graph is considered a graph where classes and its properties are nodes, those are related with their own properties and other objects through the edges.

```
Query query = db.query();
query.constrain(Person.class);
query.descend("name").constrain("Freddie Mercury");
ObjectSet result = query.execute();
```

Finally, db4o also comes with an Eclipse plugin that allows to execute queries without need of a running program:

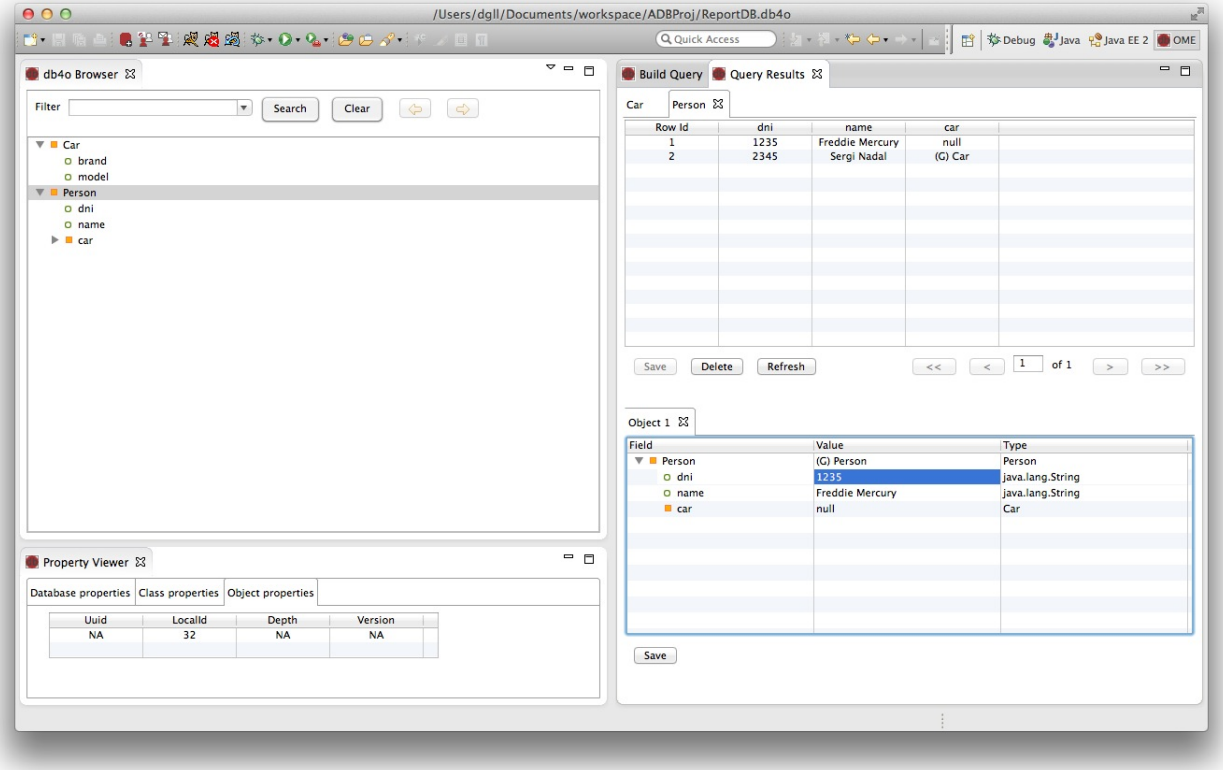

Figure 2.2: db4o Eclipse plugin

3 Comparison between LINQ and db4o

In this last chapter, we will consider the same model seen on class for LINQ. We will translate some shown queries for LINQ in all possible ways for db4o in order to compare both systems.

3.1 Model

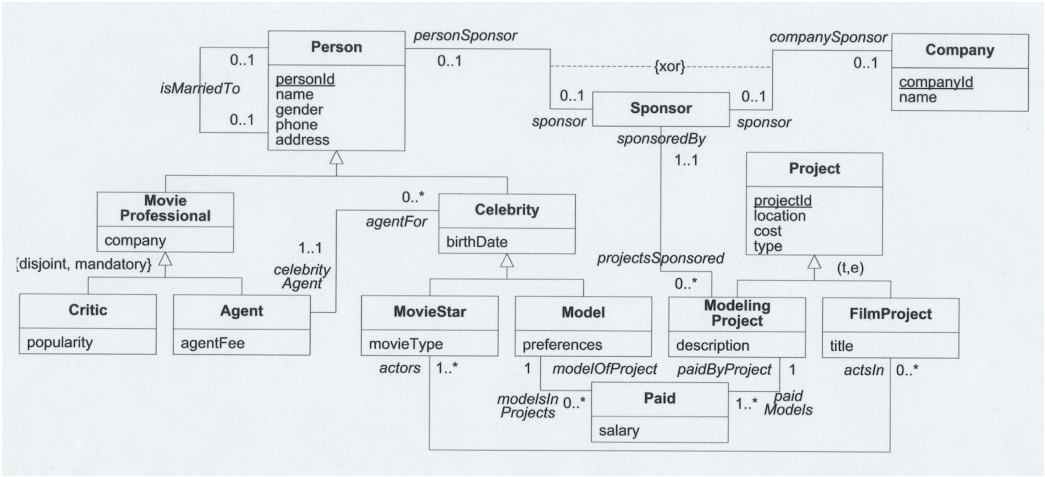

Figure 3.1: Hollywood Database model

The Java code to create the model is as follows (getter and setter operations have been omitted due to them being non-relevant):

```
1 public interface Sponsor {
2 }
```

```
3
4  public abstract class Project {
5        private int projectId;
6        private String location;
7        private int cost;
8        private String type;
9  }
10
11 public class Person implements Sponsor {
12       private int personId;
13       private String name;
14       private char gender; //'m': male; 'f': female
15       private String phone;
16       private String address;
17       private ArrayList<ModelingProject> sponseredProjects;
18       private Person marriedTo;
19 }
20
21 public class Paid {
22       private int salary;
23       private Model model;
24       private ModelingProject project;
25 }
26
27 public class MovieStar extends Celebrity {
28       private String movieType;
29       private ArrayList<FilmProject> actsIn;
30 }
31
32 public abstract class MovieProfessional extends Person {
```

```
33        private String company;
34  }
35
36  public class ModelingProject extends Project {
37        private String description;
38        private Sponsor sponsoredBy;
39        private ArrayList<Paid> models;
40  }
41
42  public class Model extends Celebrity {
43        private String preferences;
44        private ArrayList<Paid> projects;
45  }
46
47  public class FilmProject extends Project {
48        private String title;
49        private ArrayList<MovieStar> movieStars;
50  }
51
52  public class Critic extends MovieProfessional {
53        private int popularity;
54  }
55
56  public class Company implements Sponsor {
57        private int companyId;
58        private String name;
59        private ArrayList<ModelingProject> sponseredProjects;
60  }
61
62  public abstract class Celebrity extends Person {
```

```
63        private Date birthDate;
64        private Agent agent;
65  }
66
67  public class Agent extends MovieProfessional {
68        private int agentFee;
69        private ArrayList<Celebrity> agentFor;
70  }
```

3.2 Queries

All queries are written in both Native Query and SODA Query API, only few of them are written in Query By Example due to the limits stated in section 2.2.1.

- **Find the titles of the movies filmed in Phoenix**
  - *LINQ*

```
1  from f in filmProjects
2  where f.Location == "Phoenix"
3  select f.Title;
```

  - *Query By Example*

```
1  FilmProject proto = new FilmProject();
2  proto.setLocation("Phoenix");
3  ObjectSet<FilmProject> result = db.queryByExample(proto);
4  for (FilmProject o : result) {
5     System.out.println(o.getTitle());
6  }
```

  - *Native Queries*

```
1  List<FilmProject> result = db.query(new Predicate<FilmProject>() {
```

```
@Override
    public boolean match(FilmProject pro) {
        return pro.getLocation().equals("Phoenix");
    }
});
```

– *SODA Query API*

```
Query q = db.query();
q.constrain(FilmProject.class);
q.descend("location").constrain("Phoenix");
ObjectSet<FilmProject> result = q.execute();
```

In this basic query we can see the differences in the ways to query the database using each db4o's query types or LINQ. But the simplicity of this query makes it a quite poor example to see the usability of each alternative.

- **Find the actors who starred in projects filmed in Phoenix**

  – *LINQ*

```
from f in filmProjects
where f.Location == "Phoenix"
from a in f.Actors
select a.Name;
```

  – *Query By Example*

```
FilmProject proto = new FilmProject();
proto.setLocation("Phoenix");
ObjectSet<FilmProject> result = db.queryByExample(proto);
for (FilmProject o : result) {
    for (MovieStar ms : o.getMovieStars()) {
        System.out.println(ms.getName());
    }
```

```
8 }
```

– *Native Queries*

```
1  List<MovieStar> result = db.query(new Predicate<MovieStar>() {
2     @Override
3     public boolean match(MovieStar x) {
4        for (FilmProject o: x.getActsIn()) {
5           if (o.getLocation().equals("Phoenix")) {
6              return true;
7           }
8        }
9        return false;
10    }
11 });
```

– *SODA Query API*

```
1 Query q = db.query();
2 q.constrain(MovieStar.class);
3 q.descend("actsIn").descend("location").constrain("Phoenix");
4 ObjectSet<MovieStar> result = q.execute();
```

In this query we can see how each alternative manages going through relationships. In each case we basically access the attribute representing the relationship of our interest. In the NQ and SODA query, if an actor has been in two or more projects from Phoenix, he will appear only once in the result, this is a way to implement a DISTINCT clause with those methods.

- **Find celebrities who are younger than 30**

  – *LINQ*

```
1 from c in celebrities
```

```
2  where c.Age() < 30
3  select c.Name;
```

– *Native Queries*

```
1  List<Celebrity> result = db.query(new Predicate<Celebrity>() {
2     @Override
3     public boolean match(Celebrity arg0) {
4        Date d = new Date(System.currentTimeMillis());
5        d.setYear(d.getYear()-30);
6        return d.before(arg0.getBirthDate());
7     }
8  });
```

– *SODA Query API*

```
1  Query q = db.query();
2  q.constrain(Celebrity.class);
3  Date d = new Date(System.currentTimeMillis());
4  d.setYear(d.getYear()-30);
5  q.descend("birthDate").constrain(d).smaller();
6  ObjectSet<Celebrity> result = q.execute();
```

This query is an example of kind of query not possible in QBE. In QBE there is no way to express a range condition, but in NQ and SODA the way to express it is fairly simple.

- **Find the Phoenix film projects with a cost greater than $10 million**

  – *LINQ*

```
1  from phx in ( from f in filmProjects
2    where f.Location == "Phoenix" select f)
3  where phx.Cost > 10,000,000
4  select new { phx.Title, phx.Cost };
```

– *Native Queries*

```
List<FilmProject> result = db.query(new Predicate<FilmProject>() {
    @Override
    public boolean match(FilmProject arg0) {
        return arg0.getLocation().equals("Phoenix") && arg0.getCost() >
            10000000;
    }
});
System.out.println("Title Cost");
for (FilmProject o: result) {
    System.out.println(o.getTitle() + " " + o.getCost());
}
```

– *SODA Query API*

```
Query q = db.query();
q.constrain(FilmProject.class);
Constraint c = q.descend("cost").constrain(10000000).greater();
q.descend("location").constrain("Phoenix").and(c);
ObjectSet<FilmProject> result = q.execute();
System.out.println("Title cost");
for (FilmProject o: result) {
    System.out.println(o.getTitle() + " " + o.getCost());
}
```

When using NQ, expressing a Where with the form of a conjunction of conditions is done, in this case, using the logical operator && with the two conditions of the query, on the other hand, to do so with SODA, we must define another constraint from the query and then add this constraint to the other one using the operation `and()` (it would be the same for a disjunction but using the operator || or the operation `or()`, respectively).

- **Names of the celebrities that an agent manages**

  – *LINQ*

```
from a in agents
select new { agentName = a.Name,
 agentForCelebrities = (from c in a.AgentFor select c.Name) };
```

  – *Native Queries*

```
List<Agent> result = db.query(new Predicate<Agent>() {
   @Override
   public boolean match(Agent arg0) {
      return true;
   }
});
System.out.println("Agent.Name Celebrity.Name");
for (Agent o: result) {
   for (Celebrity c : o.getAgentFor()) {
      System.out.println(o.getName() + " " + c.getName());
   }
}
```

  – *SODA Query API*

```
Query q = db.query();
q.constrain(Agent.class);
ObjectSet<Agent> result = q.execute();
System.out.println("Name Celebrity");
   for (Agent o: result) {
      System.out.print(o.getName() + " ");
      for (Celebrity c: o.getAgentFor()) {
         System.out.print(c.getName() + ", ");
      }
```

```
10        System.out.println();
11    }
12 }
```

In both cases, the query itself only constraints that we want Agents, then after the query has been executed we access to the celebrities of each agent to retrieve their names. (We output the results of each way to query in two different ways, in one we output the name of the agent and the name of the celebrity for each celebrity of the agent, i.e. two rows for an agent with two celebrities; and in the other one we output the name of the agent followed by the names of his celebrities, i.e. one row for an agent for an agent with two celebrities)

- **Find the names of movie stars for which all of their film projects cost more than $20 million**
  - *LINQ*

```
1 from m in movieStars
2 where (m.ActsIn).All(fp => fp.Cost > 20000000)
3 select m.Name;
```

  - *Native Queries*

```
1  List<MovieStar> result = db.query(new Predicate<MovieStar>() {
2     @Override
3     public boolean match(MovieStar arg0) {
4        for (FilmProject p: arg0.getActsIn()) {
5           if (p.getCost() <= 20000000) {
6              return false;
7           }
8        }
9        return true;
10    }
```

```
11 });
12 System.out.println("Name");
13 for (MovieStar o: result) {
14    System.out.println(o.getName());
15 }
```

– *SODA Query API*

```
1  Query q = db.query();
2  q.constrain(MovieStar.class);
3  q.constrain(new ev6SODA());
4  ObjectSet<MovieStar> result = q.execute();
5  System.out.println("Name");
6  for (MovieStar o: result) {
7     System.out.println(o.getName());
8  }
9
10 private static class ev6SODA implements Evaluation {
11    @Override
12    public void evaluate(Candidate arg0) {
13       MovieStar ms = (MovieStar) arg0.getObject();
14       boolean res = true;
15       for (FilmProject p: ms.getActsIn()) {
16          if (p.getCost() <= 20000000) {
17             res = false;
18          }
19       }
20       arg0.include(res);
21    }
22 }
```

This query is important due to the fact that a condition must be satisfied by all the members of a relation. For NQ this is done in the `match()` method, accessing the projects of each movie star. In case any doesn't satisfy the condition, then the movie star must not be in the result. For SODA, the fact of not having an ALL-clause (similar to LINQ's one) makes the developer to implement an Evaluation class that is very similar to the match method of NQs.

- **Find the sponsors that are companies for which all of their sponsored modeling projects include at least one male model**

  – *LINQ*

```
1  from s in sponsors
2  where s.CompanySponsor ! = null &&
3    (s.ProjectsSponsored).All(mp => (mp.PaidModels).
4      Any(p => p.ModelOfProject.Gender == "M"))
5  select s.CompanySponsor.CName;
```

  – *Native Queries*

```
1  List<Company> result = db.query(new Predicate<Company>() {
2     @Override
3     public boolean match(Company arg0) {
4        for (ModelingProject mp : arg0.getSponseredProjects()) {
5           boolean bo = false;
6           for (Paid p: mp.getModels()) {
7              if (p.getModel().getGender() == 'm') {
8                 bo = true;
9              }
10          }
11          if (!bo) return false;
12       }
13       return true;
```

```
14     }
15 });
16 System.out.println("Name");
17 for (Company o: result) {
18     System.out.println(o.getName());
19 }
```

– *SODA Query API*

```
1  Query q = db.query();
2  q.constrain(Company.class);
3  q.constrain(new ev7SODA());
4  ObjectSet<Company> result = q.execute();
5  System.out.println("Name");
6  for (Company o: result) {
7     System.out.println(o.getName());
8  }
9
10 private static class ev7SODA implements Evaluation {
11    @Override
12    public void evaluate(Candidate arg0) {
13       Company c = (Company) arg0.getObject();
14       arg0.objectContainer().activate(c, 7);
15       boolean res = true;
16       for (ModelingProject mp: c.getSponseredProjects()) {
17          boolean compleix = false;
18          for (Paid p: mp.getModels()) {
19             if (p.getModel().getGender() == 'm') {
20                compleix = true;
21             }
22          }
23          if (!compleix) {
```

```
24              res = false;
25           }
26        }
27        arg0.include(res);
28     }
29  }
```

Similar example as the query before, in this case also have a condition that must be satisfied for all the objects of a relation (to have at least one male model). As in the previous example, this behavior is managed inside the `match()` or `evaluate()` method of the Evaluation class. In this query is also worthy to make notice of the line `arg0.objectContainer().activate(c, 7);` in the SODA Evaluation class, this is needed due to the fact that we will need to access objects deep in the object we just retrieved, in this case we're asking the db to activate all the objects in 7 or less relations steps of the given object, if we hadn't done this we would have found that all objects beyond the default activation level are set to null.

- **List the film projects filmed in Phoenix sorted in descending order by the cost of the project**
  - *LINQ*

```
from fp in filmProjects
where fp.Location == "Phoenix"
orderby fp.Cost descending
select new { phxTitle = fp.Title, phxCost = fp.Cost};
```

  - *Native Queries*

```
List<FilmProject> result = db.query(new Predicate<FilmProject>() {
   @Override
   public boolean match(FilmProject arg0) {
      return arg0.getLocation().equals("Phoenix");
```

```
5     }
6  });
7  ArrayList<FilmProject> res = new ArrayList<FilmProject>();
8  for (FilmProject op: result) {
9     res.add(op);
10 }
11 System.out.println("Title Cost");
12 Collections.sort(res, new Comparator<FilmProject>() {
13    @Override
14    public int compare(FilmProject o1, FilmProject o2) {
15       if (o1.getCost() < o2.getCost()) {
16          return 1;
17       }
18       return -1;
19    }
20 });
21 for (FilmProject o: res) {
22    System.out.println(o.getTitle() + " " + o.getCost());
23 }
```

– *SODA Query API*

```
1  Query q = db.query();
2  q.constrain(FilmProject.class);
3  q.descend("location").constrain("Phoenix");
4  q.descend("cost").orderDescending();
5  ObjectSet<FilmProject> result = q.execute();
6  System.out.println("Title cost");
7  for (FilmProject o: result) {
8     System.out.println(o.getTitle() + " " + o.getCost());
9  }
```

In this example we are asked to sort the query's result by cost. For NQs, there's actually no native support for sorting results, hence we must store the result in an array and use the Java method `Collections.sort()` to have the result sorted. In SODA though, there is an operation for sorting, making this step way easier than with NQ.

- **List of movie stars in alphabetical order for each film project filmed in Phoenix**
  - *LINQ*

```
from fp in filmProjects
where fp.Location == "Phoenix"
orderby fp.Title ascending
select new { phxTitle = fp.Title, phxMovieStars =
  ( from ms in fp.Actors orderby ms.Name select ms.Name ) };
```

  - *Native Queries*

```
List<FilmProject> result = db.query(new Predicate<FilmProject>() {
   @Override
   public boolean match(FilmProject arg0) {
      return arg0.getLocation().equals("Phoenix");
   }
});
ArrayList<FilmProject> res = new ArrayList<FilmProject>();
for (FilmProject op: result) {
   res.add(op);
}
System.out.println("Title Names");
Collections.sort(res, new Comparator<FilmProject>() {
   @Override
   public int compare(FilmProject o1, FilmProject o2) {
```

```
15        return o1.getTitle().compareTo(o2.getTitle());
16     }
17  });
18  for (FilmProject o: res) {
19     ArrayList<MovieStar> ress = o.getMovieStars();
20     Collections.sort(ress, new Comparator<MovieStar>() {
21        @Override
22        public int compare(MovieStar o1, MovieStar o2) {
23           return o1.getName().compareTo(o2.getName());
24        }
25     });
26     for (MovieStar m: ress) {
27        System.out.println(o.getTitle() + " " + m.getName());
28     }
29  }
```

– *SODA Query API*

```
1  Query q = db.query();
2  q.constrain(FilmProject.class);
3  q.descend("location").constrain("Phoenix");
4  q.descend("title").orderAscending();
5  q.descend("movieStars").descend("name").orderAscending();
6  ObjectSet<FilmProject> result = q.execute();
7  System.out.println("Title Names");
8  for (FilmProject o: result) {
9     System.out.print(o.getTitle() + " ");
10    for (MovieStar m: o.getMovieStars()) {
11       System.out.print(m.getName() + ", ");
12    }
13    System.out.println();
14 }
```

Another example of sorting results, in this case the result is sorted twice, first by title of the film projects and then by the name of the actors. As we have seen in the previous query, this is done in an easier way with SODA than with NQ.

- **Finds the highest paid Phoenix model**
  - *LINQ*

```
( from m in models
from p in m.ModelsInProjects
where p.PaidByProject.Location == "Phoenix"
orderby p.Salary ascending
select new {modelName = m.Name, modelSalary = p.Salary}).Last();
```

  - *Native Queries*

```
List<Paid> result = db.query(new Predicate<Paid>() {
   @Override
   public boolean match(Paid arg0) {
      return arg0.getProject().getLocation().equals("Phoenix");
   }
});
ArrayList<Paid> res = new ArrayList<Paid>();
for (Paid op: result) {
   res.add(op);
}
System.out.println("Name Salary");
Collections.sort(res, new Comparator<Paid>() {
   @Override
   public int compare(Paid o1, Paid o2) {
      if (o1.getSalary() < o2.getSalary()) return 1;
      return -1;
   }
```

```
18 });
19 if (!res.isEmpty()) {
20    System.out.println(res.get(0).getModel().getName() + " " + res.get(0).
          getSalary());
21 }
```

– *SODA Query API*

```
1 Query q = db.query();
2 q.constrain(Paid.class);
3 q.descend("project").descend("location").constrain("Phoenix");
4 q.descend("salary").orderDescending();
5 ObjectSet<Paid> result = q.execute();
6 System.out.println("Name Salary");
7 if (!result.isEmpty()) {
8    System.out.println(result.get(0).getModel().getName() + " " + result.get
          (0).getSalary());
9 }
```

The LINQ version of the query uses the method `Last()` to obtain the last row in the result set (after sorting it by salary), obtaining the model with the highest salary that has worked in Phoenix. For NQ and SODA there's no native operation to achieve that, so we sort the result and then get the first one (we ordered descending instead of ascending, hence we have to get the first one instead of the last one).

- **Determine the number of films for movie stars**

  – *LINQ*

```
1 from m in movieStars
2 orderby m.Name
3 select new {name = m.Name, filmCount = m.ActsIn.Count()};
```

  – *Native Queries*

```
List<MovieStar> result = db.query(new Predicate<MovieStar>() {
   @Override
   public boolean match(MovieStar arg0) {
      return true;
   }
});
ArrayList<MovieStar> res = new ArrayList<MovieStar>();
for (MovieStar op: result) {
   res.add(op);
}
System.out.println("Name Count");
Collections.sort(res, new Comparator<MovieStar>() {
   @Override
   public int compare(MovieStar o1, MovieStar o2) {
      return o1.getName().compareTo(o2.getName());
   }
});
for (MovieStar o: res) {
   System.out.println(o.getName() + " " + o.getActsIn().size());
}
```

– *SODA Query API*

```
Query q = db.query();
q.constrain(MovieStar.class);
q.descend("name").orderDescending();
ObjectSet<MovieStar> result = q.execute();
System.out.println("Name FilmCount");
for (MovieStar o: result) {
   System.out.println(o.getName() + " " + o.getActsIn().size());
}
```

Another translation from LINQ to NQ/SODA, in this case we use the `size()` operation on the array of Films to count the number of films in which each movie star has been. Another way to do it for NQ would have been to implement a subclass of Predicate and keep a map with the number of films of each movie star, we will see this in the following queries.

- **Find the total salary for a model**
  - *LINQ*

```
from m in models
select new { name = m.Name, salaryTotal =
  ( from p in m.ModelsInProjects select p.Salary ).Sum()};
```

  - *Native Queries*

```
List<Model> result = db.query(new Predicate<Model>() {
   @Override
   public boolean match(Model arg0) {
      return true;
   }
});
System.out.println("Name SalaryTotal");
for (Model o: result) {
   int sum = 0;
   for (Paid p: o.getProjects()) {
      sum += p.getSalary();
   }
   System.out.println(o.getName() + " " + sum);
}
```

  - *SODA Query API*

```
Query q = db.query();
```

```
2  q.constrain(Model.class);
3  ObjectSet<Model> result = q.execute();
4  System.out.println("Name SalaryTotal");
5  for (Model o: result) {
6     int sum = 0;
7     for (Paid p: o.getProjects()) {
8        sum += p.getSalary();
9     }
10    System.out.println(o.getName() + " " + sum);
11 }
```

Similar query as the previous one but in this case we want the sum of the salary. This is done in a similar way as the count in the last query. As in the previous query, this could be done keeping a map with the sum of salary for each model.

- **Group film projects by their location and give the location, count of the number of films associated with that location, and the group of films**
  - *LINQ*

```
1  var filmsByLocation =
2    from f in filmProjects
3    group f by f.Location;
4  from locFilms in filmsByLocation
5  orderby locFilms.Key
6  select new { location = locFilms.Key, numberOfFilms = locFilms.Count(),
7    films = locFilms };
```

  - *Native Queries*

```
1  Query13Predicate predicate = new Query13Predicate();
2  List<FilmProject> result = db.query(bla);
3  System.out.println("Location Count Films");
4  for (Map.Entry<String, Integer> ce: predicate.countTable.entrySet()) {
```

```
5     System.out.print(ce.getKey() + " " + ce.getValue() + " ");
6     for (FilmProject o: predicate.filmTable.get(ce.getKey())) {
7        System.out.print(o.getTitle() + ", ");
8     }
9     System.out.println();
10 }
11
12 private static class Query13Predicate extends Predicate<FilmProject> {
13    public HashMap<String, Integer> countTable = new HashMap<String, Integer
          >();
14    public HashMap<String, ArrayList<FilmProject>> filmTable = new HashMap<
          String, ArrayList<FilmProject>>();
15    @Override
16    public boolean match(FilmProject arg0) {
17       if (countTable.containsKey(arg0.getLocation())) {
18          countTable.put(arg0.getLocation(), countTable.get(arg0.getLocation
                ()) + 1);
19          filmTable.get(arg0.getLocation()).add(arg0);
20       } else {
21          countTable.put(arg0.getLocation(), 1);
22          ArrayList<FilmProject> a = new ArrayList<FilmProject>();
23          a.add(arg0);
24          filmTable.put(arg0.getLocation(), a);
25       }
26    return true;
27 }
```

– *SODA Query API*

```
1 Query q = db.query();
2 q.constrain(FilmProject.class);
3 q.descend("location").orderDescending();
```

```
4  ObjectSet<FilmProject> result = q.execute();
5  System.out.println("Location numberOfFilms Films");
6  HashMap<String, ArrayList<FilmProject>> map = new HashMap<String, ArrayList<
       FilmProject>>();
7  for (FilmProject o: result) {
8     if (map.containsKey(o.getLocation())){
9        map.get(o.getLocation()).add(o);
10    } else {
11       ArrayList<FilmProject> set = new ArrayList<FilmProject>();
12       set.add(o);
13       map.put(o.getLocation(), set);
14    }
15 }
16 for (Entry<String, ArrayList<FilmProject>> entry : map.entrySet()) {
17    System.out.print(entry.getKey() + " " + entry.getValue().size() + " ");
18    for (FilmProject o: entry.getValue()) {
19       System.out.print(o.getTitle() + ", ");
20    }
21    System.out.println();
22 }
```

This is the first query using the group by clause. While the grouping is quite simple in LINQ, it becomes a more tedious task in db4o. Both in NQ and SODA (and it would be the same for QBE) we must keep a map representing the grouping (the key is the grouping attribute and the value is the elements in that group), this is done with a subclass of Predicate for the native queries and done after the execution of the query for SODA (we skipped the sorting code in NQ in order to keep the code simple for the non-relevant parts of the query, we have already seen how to sort with NQ). This query can be done in two different ways: with two maps, one for the counting and one for the actual elements in the group (the NQ

version), or with only one map having the elements in the group and executing the `size()` method to count them (SODA version).

- **Group models by their agent, returning for each agent that represents more than three models, the count and names of the models in the group**

  – *LINQ*

```
from m in models
group m by m.CelebrityAgent into agentGroup
where agentGroup.Count() > 3
select new { agentName = agentGroup.Key.Name, modelCount = agentGroup.Count
    (),
 modelNames = (from mg in agentGroup select mg.Name) };
```

  – *Native Queries*

```
Query14Predicate predicate = new Query14Predicate();
List<Model> result = db.query(predicate);
System.out.println("Agent Count Names");
for (Map.Entry<String, ArrayList<Model>> ce: predicate.table.entrySet()) {
   if (ce.getValue().size() > 3) {
      System.out.print(ce.getKey() + " " + ce.getValue().size() + " ");
      for (Model o: ce.getValue()) {
         System.out.print(o.getName() + ", ");
      }
      System.out.println();
   }
}

private static class Query14Predicate extends Predicate<Model> {
   public HashMap<String, ArrayList<Model>> table = new HashMap<String,
       ArrayList<Model>>();
```

```
16     @Override
17     public boolean match(Model arg0) {
18        if (table.containsKey(arg0.getAgent().getName())) {
19           table.get(arg0.getAgent().getName()).add(arg0);
20        } else {
21           ArrayList<Model> a = new ArrayList<Model>();
22           a.add(arg0);
23           table.put(arg0.getAgent().getName(), a);
24        }
25        return true;
26     }
27  }
```

– *SODA Query API*

```
1  Query q = db.query();
2  q.constrain(Model.class);
3  ObjectSet<Model> result = q.execute();
4  System.out.println("Name modelCount modelNames");
5  HashMap<String, ArrayList<Model>> map = new HashMap<String, ArrayList<Model
       >>();
6  for (Model o: result) {
7     if (map.containsKey(o.getAgent().getName())){
8        map.get(o.getAgent().getName()).add(o);
9     } else {
10       ArrayList<Model> set = new ArrayList<Model>();
11       set.add(o);
12       map.put(o.getAgent().getName(), set);
13    }
14 }
15 for (Entry<String, ArrayList<Model>> entry : map.entrySet()) {
16    if (entry.getValue().size() > 3) {
```

```
17        System.out.print(entry.getKey() + " " + entry.getValue().size() + " ")
              ;
18        for (Model o: entry.getValue()) {
19           System.out.print(o.getName() + ", ");
20        }
21        System.out.println();
22     }
23  }
```

A LINQ query expressing a restriction on the subgroups (HAVING in SQL). As in grouping, there is no way to express this either with SODA nor NQ, so we must check each entry in the map representing the grouping in order to ignore those that don't have more than 3 models.

- **Define a variable loc and a query the films filmed at the loc**
  - *LINQ*

```
1  var loc = "Phoenix";
2  var filmsAtLocation =
3    from f in filmProjects
4    where f.Location == loc
5    select f;
```

  - *Native Queries*

```
1  final String loc = new String("Phoenix");
2  List<FilmProject> result = db.query(new Predicate<FilmProject>() {
3     @Override
4     public boolean match(FilmProject arg0) {
5        return arg0.getLocation().equals(loc);
6     }
7  });
8  System.out.println("Title");
```

```
for (FilmProject o: result) {
   System.out.println(o.getTitle());
}
```

– *SODA Query API*

```
String loc = new String("Phoenix");
Query q = db.query();
q.constrain(FilmProject.class);
q.descend("location").constrain(loc);
ObjectSet<FilmProject> result = q.execute();
System.out.println("Title");
for (FilmProject o: result) {
   System.out.println(o.getTitle());
}
```

This is an example of the use of variables inside a query. It's done quite similarly in LINQ, NQ and SODA.

3.3 Conclusions

It is hard to compare querying on db4o and LINQ, this is because the first one is an OODB while LINQ is ORDB, but during the implementation of the db4o queries there are indeed some differences we have noticed:

- **Simplicity for simple queries** When all you need is to do a simple query, the best way to go is the db4o one. The fact of being able to query the database just creating an instance of what you're looking for is quite a good point in favor of db4o.

- **Tricky parts with complex queries** When facing a complex query (without taking into account groups by and aggregations), it can be a little bit tricky with db4o due to having to code some extra procedure in order to check, for example, that all the members of a collection satisfy some condition.

- **Troublesome grouping** This problem is not an only-db4o problem but a problem of the OODBs. These databases are really weak when it comes to group instances and aggregate some attributes. Hence, a group by which is easily done with LINQ, with db4o you have to do it yourself by having a Map for each output group, and this can really tricky when grouping by multiples attributes (a Map of Maps).

  Hence our conclusion of this project is that, in a general case, there could be of course some use cases that is the other way around, it's better to use LINQ when you need queries a little bit more complex than just simple filters. But, if you don't need to group your data, db4o is definitely a better option.

4 Bibliography

4.1 Papers

[1] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, Klaus Dittrich, Stanley Zdonik. *The Object-Oriented Database System Manifesto.*

[2] Michael Stonebraker, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein, David Beech. *Third-Generation Database System Manifesto.*

[3] Versant Corporation. *db4o 8.0 tutorial.*

4.2 Courses

[4] Alberto Abelló, Oscar Romero. *Object-Oriented and Object-Relational Databases.* Part of Concepts for Specialized Databases course at FIB-UPC.

[5] Anna Queralt. *Object Databases for BigData Sharing.* BSC - Barcelona Supercomputing Center.

[6] Esteban Zimányi. *Object Databases.* Part of INFOH415 Advanced Databases courses at ULB.

4.3 Websites

[7] Service Architecture, *Impedance Mismatch When Mapping from a Relational Database*. Available at: http://www.service-architecture.com/articles/object-oriented-databases/impedance_mismatch.html

[8] Versant, *db4o blogs*. Available at: http://community.versant.com/Blogs/db4o/tabid/197/tag/db4o/Default.aspx