Object-oriented Databases
&
db4o

by

Annemarie Burger (479207)
&
Pinar Turkyilmaz (476728)

INFO-H-415: Advanced Databases
prof. Esteban Zimányi

December 17, 2018

UNIVERSITÉ LIBRE DE BRUXELLES ULB

Table of Contents

Annemarie Burger (479207), Pinar Turkyilmaz (476728)

Introduction

Object-oriented databases have been in use for quite some time, but only when db4o - database for objects - was developed, it became a more used database management system choice. In this report we will discuss the history and applications of an object-oriented database management system (OODBMS), how they compare to relational databases (RDB) and we will deal extensively with db4o; how it works, pros and cons, and we will compare its queries to SQL queries.

Object-oriented Databases

Object-oriented databases have been in and out of fashion. The development of db4o had a big influence on this, but also the history and popularity of relational (object-oriented) databases has to be considered.

History

In 1995, Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik publish an paper called 'The Object-Oriented Database System Manifesto' in which they define what an OODBMS should be like. They describe the main features and characteristics which they have split in three groups.

The first group is made up from thirteen mandatory features: five to define the system as a database management system, and eight additional ones to also define it as an object-oriented system. In the paper they wrote commandments and a brief explanation to each of the features, the first five commandments respectively are:

- Persistence: Thou shalt remember thy data
- Secondary storage management: Thou shalt manage very large databases
- Concurrency: Thou shalt accept concurrent users
- Recovery: Thou shalt recover from hardware and software failures:
- Ad Hoc Query Facility: Thou shalt have a simple way of querying data

Especially the last of these requirements has been problematic for OODBMSs, since there is no standardized query language for them as SQL is for relational database management systems. The Object Data Management Group (ODMG) created the Object Query Language

(OQL), which was very close related to SQL-92, but they disbanded the group and abandoned their efforts in 2001. (ODBMS.org II., retrieved 2018)

The eight mandatory features named to define the system as an object-oriented system are:

- Complex objects: Thou shalt support complex objects
- Object identity: Thou shalt support object identity
- Encapsulation: Thou shalt encapsulate thine objects
- Types and Classes: Thou shalt support types or classes
- Class or Type Hierarchies: Thine classes or types shalt inherit from their ancestors
- Overriding, overloading and late binding: Thou shalt not bind prematurely
- Computational completeness: Thou shalt be computationally complete
- Extensibility: Thou shalt be extensible

All of which are pretty straight-forward.

The paper also gives suggestions on optional features, which they call 'the goodies', since they "clearly improve the system, but (...) are not mandatory to make it an object-oriented database system." (Atkinson et al., 1995). Among these are multiple inheritance, type checking, and design transaction management. The final group of features the paper discusses are the open choices the system designer has to make. They include the type and representation of the system, and the uniformity.

This paper was very important in defining OODBMS, but not many ended up being made or widely used. Relational databases were and are far more popular, even though OODBMS are a better fit for object-oriented programming languages. ODBMS.org (Retrieved 2018) names a few reasons for this: "high switching cost, the inclusion of object-oriented features in RDBMS to make them ORDBMS, and the emergence of object-relational mappers (ORMs)". They are still in use though, but mostly as a complement, not a replacement for RDBMS, and they are currently enjoying a boost in popularity fueled by the open-source community. They "found their place as embeddable persistence solutions in devices, on clients, in packaged software, in real-time control systems, and to power websites." (ODBMS.org I., retrieved 2018)

Relational databases are what we call second-generation databases, and by far the most popular ones. However, with the increasing dominance of object-oriented languages as Java

it becomes more important to avoid the impedance mismatch. OODBMS make data more easily accessible, which results in greater developer productivity. They are third-generation databases, sometimes referred to as post-relational models. Because of the strong dominance of relational databases, it's been very hard for third-generation systems to gain market share, but they are still growing steadily. (Paterson et al., 2006).

Moreover, RDBMS are not performing very well when it comes to processing large amounts of complex data, such as image collections, video stream collections etc.. The increased number of the applications which require to handle these complex data motivates developers to use object-oriented databases systems. Nowadays they are being used in areas such as Computer Integrated Manufacturing (CIM), Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), Computer Aided Software Engineering (CASE) (Saxena & Pratap, 2013). These applications use object-oriented databases to handle complex graphical data.

On the other hand, as Bagui, S. (2003) puts it, there are some cons of OODBMS, such as lack of standard query algebra which effects the query optimization. Also, most OODB do not support authorization and this increases the security concerns with OOBDs. Some other features such as constraints with UNIQUE and NULL, and triggers, are not supported by OODBMS.

db4o

Development for db4o started in 2000 by the company Actian, which started shipping it in 2001 and promoting it commercially in 2004. It was by no means the first OODBMS, since they “have been available commercially since the early 1990s, but have not had a great deal of impact outside niche markets.” (Paterson et al., 2006). This is partly because they did not have a standardized query language or data definition, and partly because they were non-native, which means that objects were not stored as the original objects. The latter is the reason why db4o was such a game changer, since it worked with a native interface.

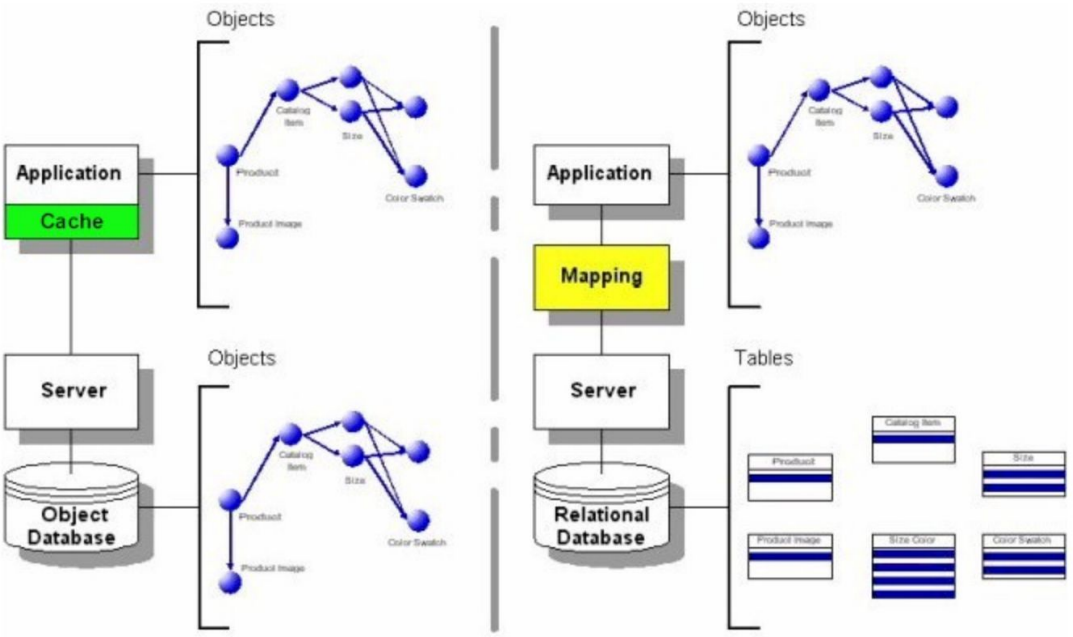
How it works

db4o is a non-relational embedded DB specifically focused on persistence. It is not string-based - as for example SQL -, but stores and queries natively, which in this case means by just using Java or .NET. There is no context switch between programming and

API language. This is mainly what makes it so different from RDB and older OODBMS; it is pure native. Objects are stored directly as the object, without chopping them up or changing their characteristics. This is intuitively more logical because if you are using SQL for objects, you basically need to disassemble them to fit them into tables. “Digital technology consultant Esther Dyson put it another way:

> *Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car.*

“ (Paterson et al., 2006). This disassembling brings all kinds of problems, most importantly the increased chance on impedance mismatch, since you have to correctly map all parts to each other. In the image below the difference between a native object database as db4o and relational database gets further explained. The impedance mismatch can occur during the mapping step, which can be completely skipped thanks to db4o’s native interface.

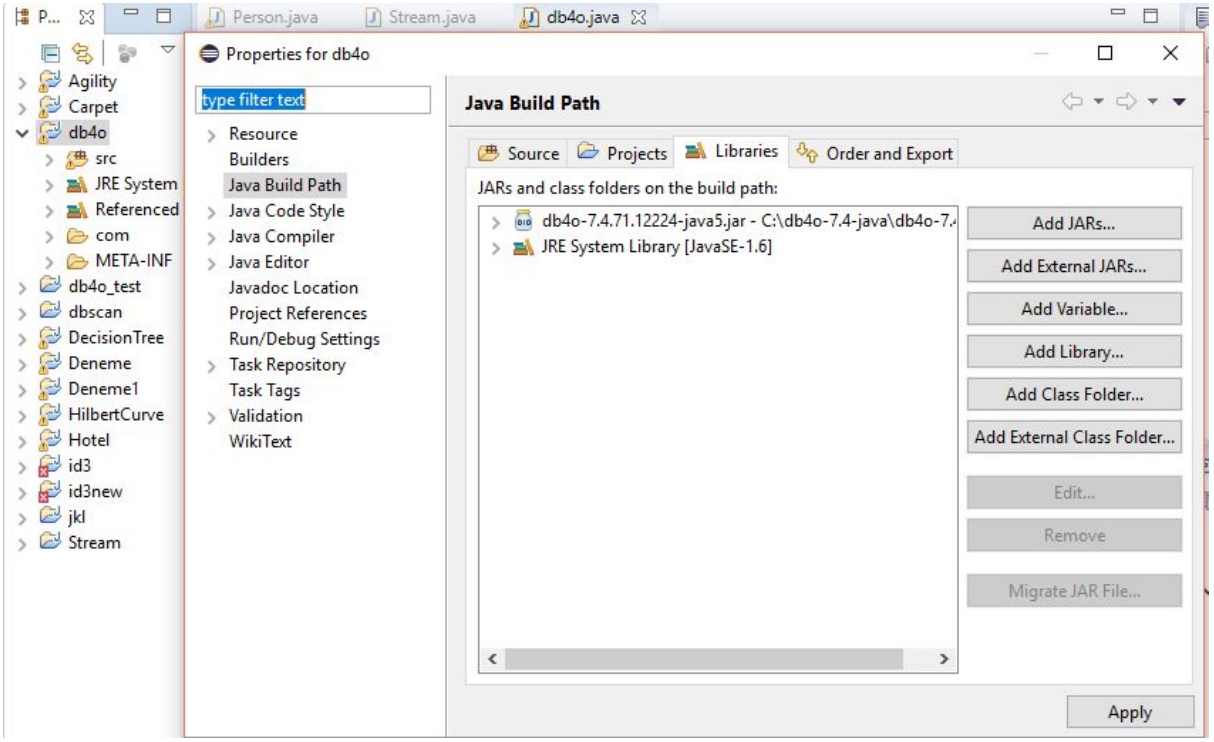


Source: https://medium.com/@gp_pulipaka/db4o-object-oriented-database-479934899b86

How to Build db4o

The version that we used is db4o 7.4-java. The Java version we use is Java 8. db4o can be downloaded from the internet very easily.

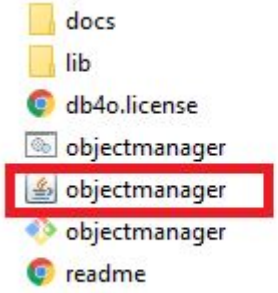
To be able to use db4o in Eclipse, simply add the appropriate .jar file under /lib/ folder of the project which you work with db4o and add db4o to your project as a library.



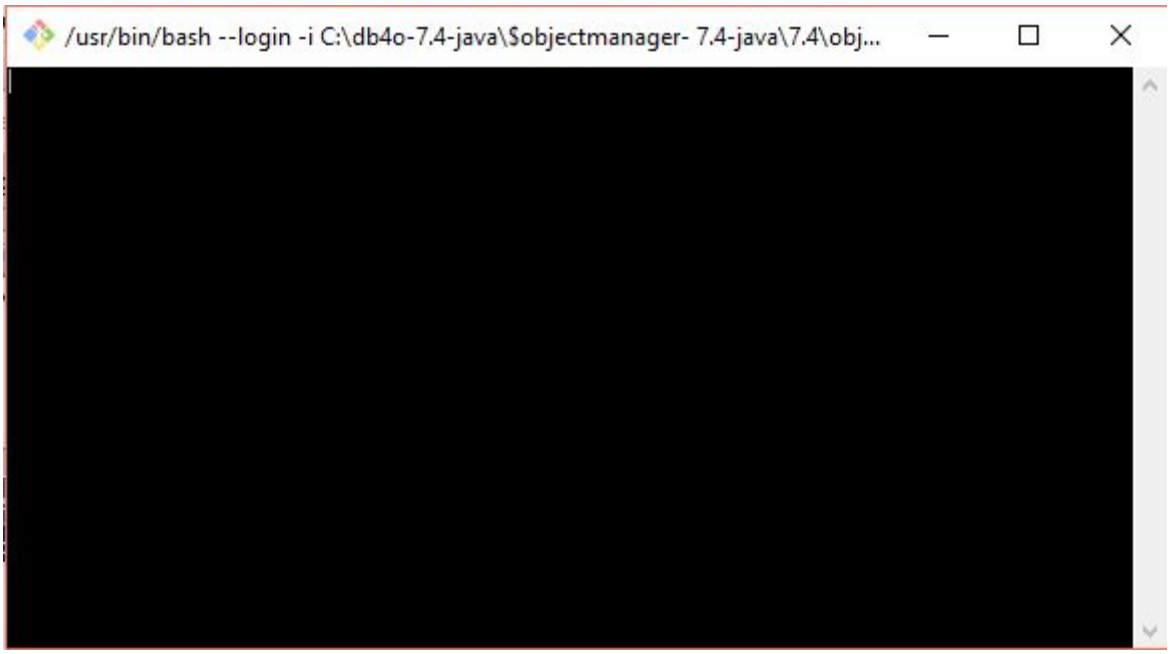
Object Manager Enterprise (OME)

Object Manager Enterprise is a very simple graphical interface of the database. It is possible to download it on the internet. Note that it should be in the same version with db4o.

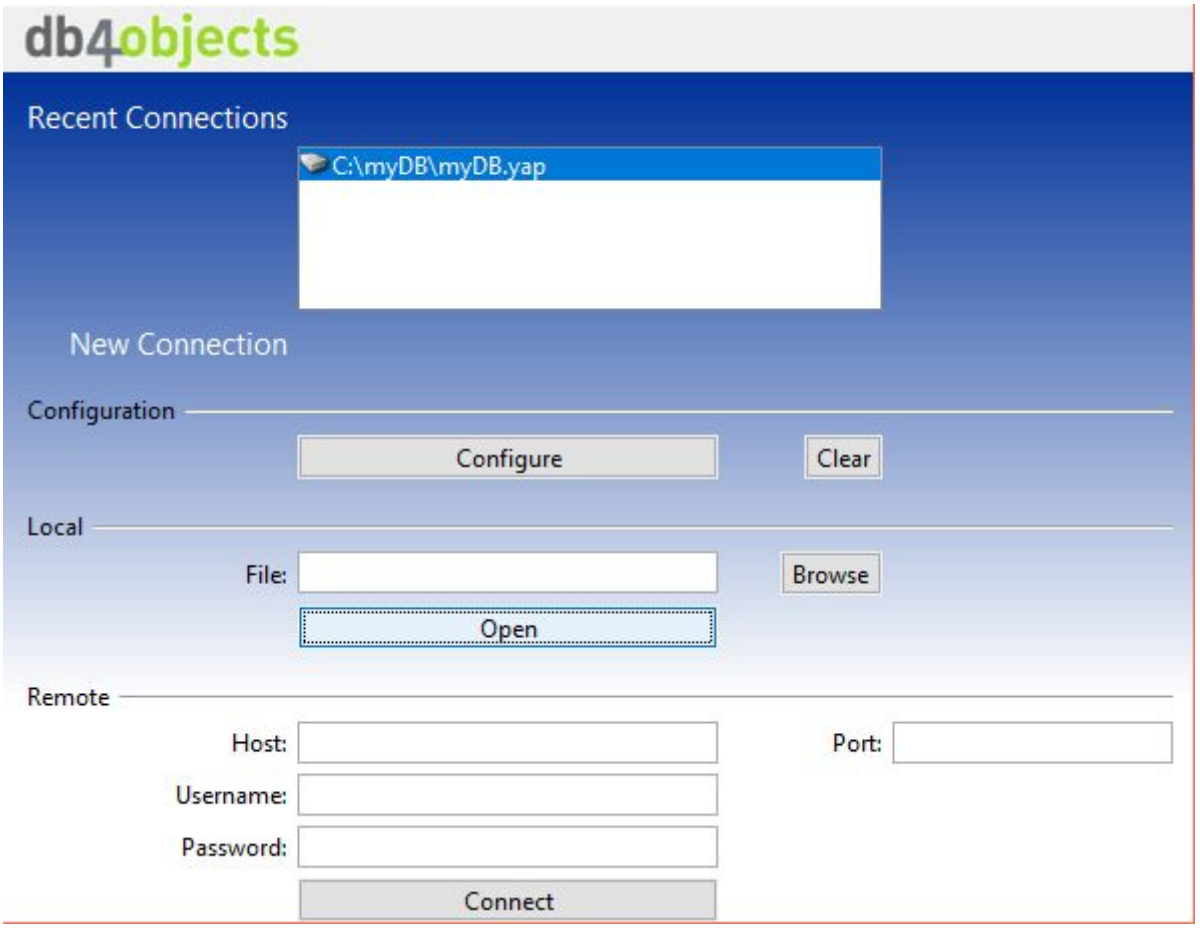
It is downloaded in zip file. To build it, just unzip the file in any location you want and run the .jar file.



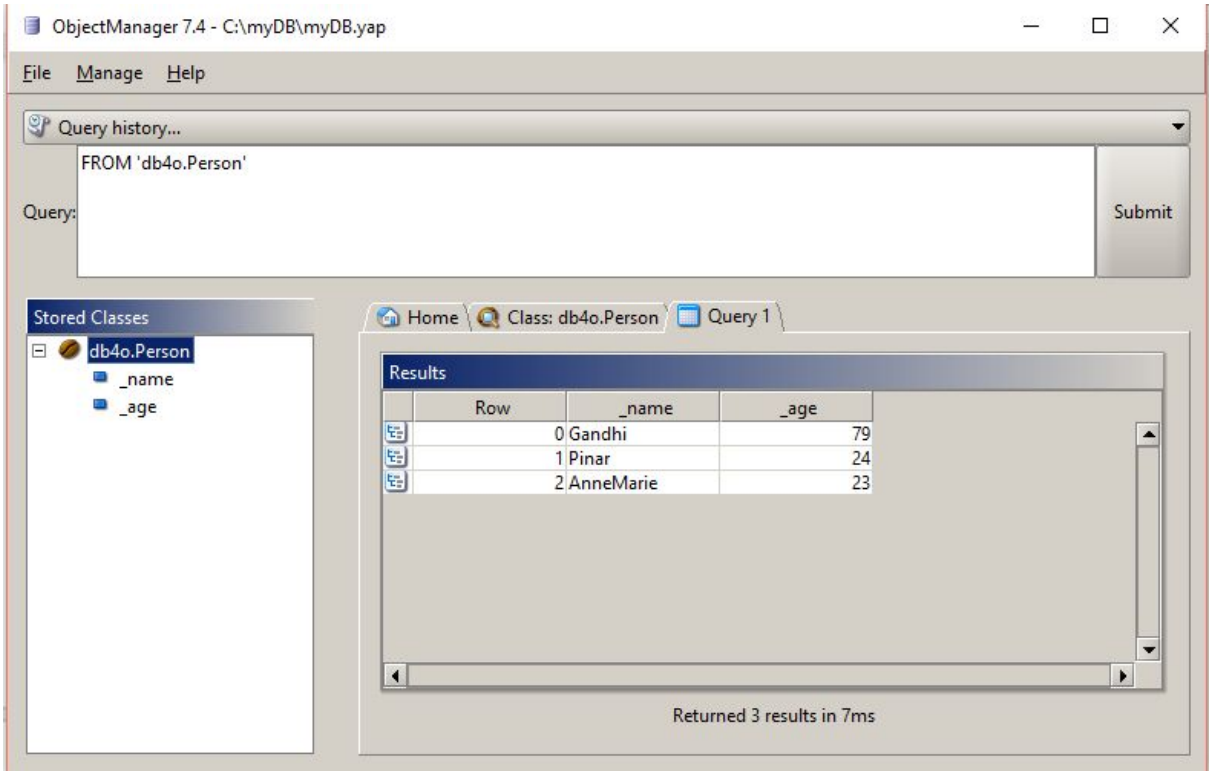
When you run objectmanager.jar file, first an empty command line will appear.



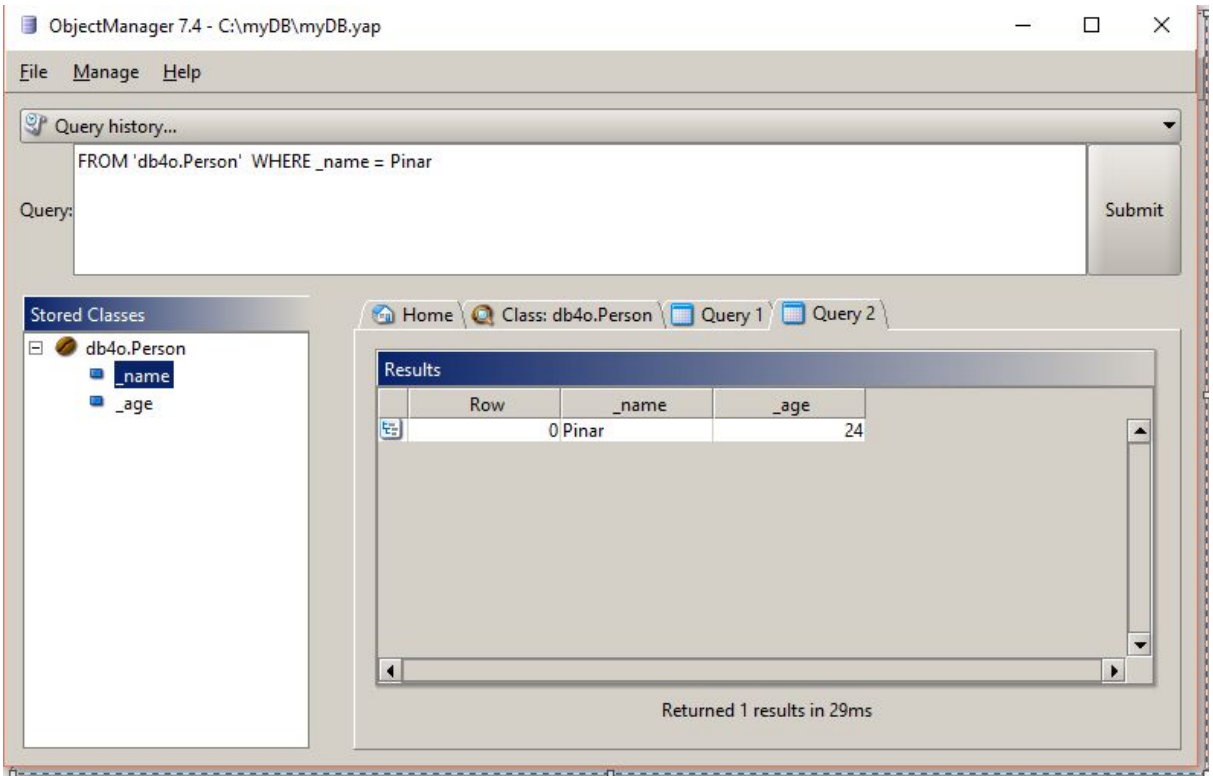
There is no need to interact with this screen. Just wait for the main window to be opened.



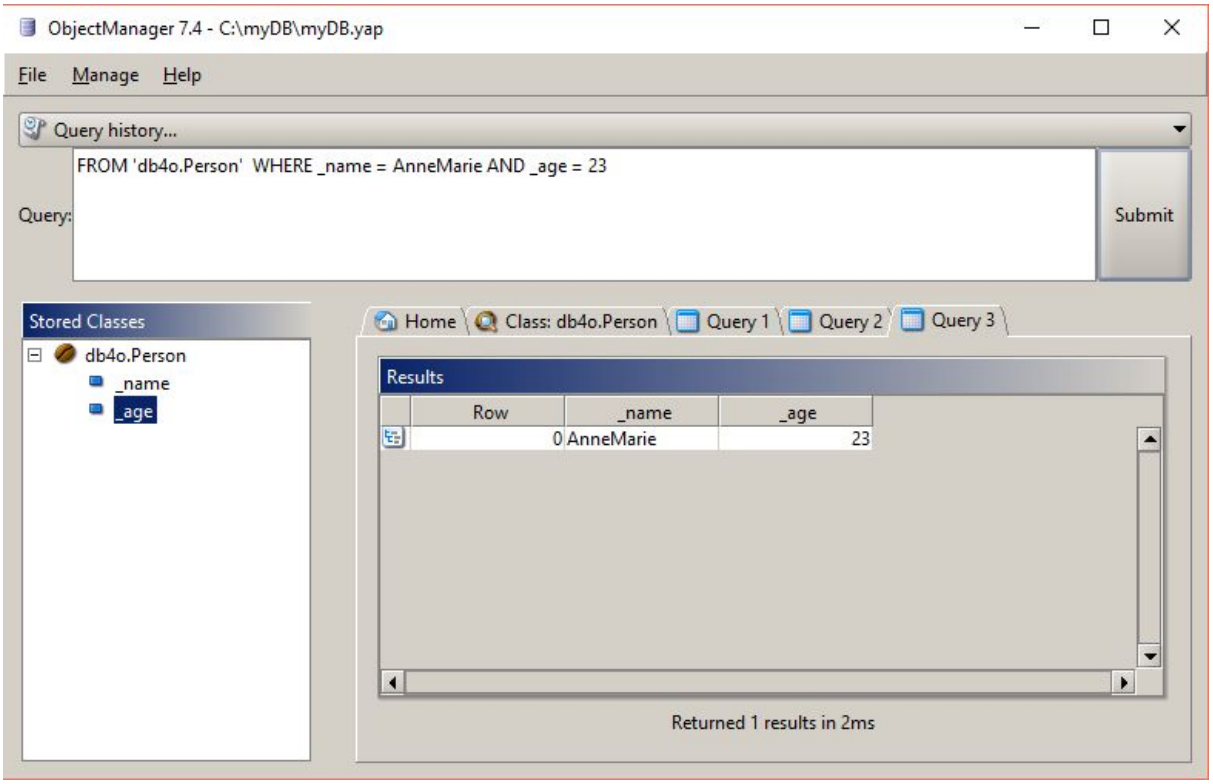
When the main screen appears, just click “Browse” button, and select the location of your database.



When you click on the db4o.*ClassName* it is possible to see the data in it. OME also gives an opportunity to create queries automatically. In the following example, when“_name” attribute is clicked, in the Query section “WHERE _name ?” will appear. Just write what you want to retrieve and the result will appear in the result section.



Also it is possible to expand the queries in the same way with the previous example.Just double click the attribute and it will be added to the query.



Queries

db4o queries are quite different from SQL queries. For example, if you want to get a specific manager and all employees it is associated with, you need two queries in SQL; one for the manager and one for its employees. In an object oriented database you only need one query since it can also right away return all the associated objects, even if they are only connected through traversing relationships. This is a very powerful tool and should be handled with care, especially when we are using very deep structures or circular references. Another way of handling it is by controlling the activation depth, that limits the amount of objects that can be returned. One should also mind the fact that queries are only used in db4o to return data. Separate methods are used for inserting, updating and deleting objects in the database. We will talk about those as well in the comparison part of this code.

db4o has three different options for queries: Query By Example (QBE), Native Queries (NQ) and SODA Query API. We will discuss them here in theory and in a more practical way in the comparison part of this report.

1: Query by example

This is the most basic way of querying and does not come close to the power of SQL since it is so simple. The idea is that you create an object with the search field value you want specified while the others are left with null or zero values, and ask the database to give you all matching objects. The query returns a collection with all the objects that match the attribute pattern you provided. The disadvantages of this method of querying is that you can only use if for very basic queries, and can not use it for ranges or more complex queries. (Paterson et al., 2006).

2: SODA Query API

SODA stands for Simple Object Data Access, and is based on the notion of a query graph. In this graph you can specify the type of objects you want and use method calls to “descend” - as db4o calls it - to objects with certain constraints. These types of queries are very fast, but quite foreign if you are very used to for example SQL, since it is based on a totally different way of thinking about queries. Because of this, it is probably the least used query method in db4o, and it is also possible as a developer to get by with just the other two options. The native queries are most recommended for db4o and while executing them

queries db4o tries to convert them into SODA queries since this is the most efficient method. (Paterson et al., 2006)

3: Native Queries

The thing that made people so excited about db4o, it's the new, cool thing, if you will, and the primary query mechanism of db4o. By using native queries, the query mechanism is completely integrated in the programming language, since it does not use declarative code. As Paterson et al., (2006) put it: "you write your application, and then you can plug in a different kind of database by changing only one line of code.(...) Native queries offer the potential to (...) provide type-safe querying of any kind of database and also of in-memory objects. This integration of querying and language offers great potential for the future." db4o will always try to optimize a query, which sometimes means a native query will get translated into a SODA query for faster results. You can express a native query in any C# or Java code that returns a Boolean. The system applies your method to all objects stored and the collection of matching objects is returned. The great advantage is that errors can be detected by your compiler because the query is native with the rest of the code. This avoids runtime errors. (Paterson et al., 2006)

Applications

As Paterson et al., (2006) puts it: "db4o is most likely to be used in applications where there is no legacy data, which is to say no existing data architectures to integrate. This is primarily the case with applications that run on clients or on middleware. Unlike most other DBMS, db4o is not built as a server system but as a library." This is one of the main things that make db4o very useful, because it makes db4o to have very small memory footprint. This feature makes db4o to be used in the applications that need to store data, but are more focused on the functionality of the application then on having full stand-alone flexibility with the data. This does not mean that data is not important, rather that data will be used in a predefined way and not afterwards for dynamic data analysis etc. With this characteristic db4o becomes very ideal for mobile, disconnected or embedded devices. This is the case for mobile handheld applications or for industrial device solutions. Also, it is possible to use db4o as database where 'heavy' DBMS are not usable, and even in some cases where (R)DBMS were considered as traditional, db4o might be an alternative choice. (Hauser, P.(n.d.))

Below some of the projects that were developed using db4o.(Paterson et al., 2006)

- Used within a high-speed train control system, developed by Indra Sistemas, in Spain's AVE rail network. No other system was able to handle the incredible load of processing over 200,000 heterogeneous objects per second.
- Used to control complex, high-speed packaging robots built by Bosch Packaging Technology Group. db4o can facilitate the complex object models required by their high-performance packaging robots with ease.
- Used to improve eye health for babies in Clarity Medical Systems' Retcam II product, which provides state-of-the-art wide-field pediatric retinal imaging for infant eye screening.
- Selected by Boeing for the P-8A Multi-Mission Maritime Aircraft, a long-range anti-submarine warfare, anti-surface warfare, intelligence, surveillance, and reconnaissance aircraft for the U.S. Navy.
- Used to replace legacy databases in LoanMaster (hand-held software designed for the home credit industry) and Mobilize Van (a route accounting distribution management system) redeveloped by Eastern Data Systems.
- Used by Mandala IT in consumer software products for mass-market cell phones.
- Indian Postal Services has been developed by using db4o and Db4o solves the problem of impedance mismatch and making the development of database model much simpler. (Saxena & Pratap (2012))

Pros & Cons

As Paterson et al., (2006) puts it: "[db4o's] key features are performance, compactness, zero administration, simplicity, and the unique ability to store native Java or .NET objects, providing cross-platform portability. Objects are stored exactly as they are—there is no need for a data definition language. Zero administration is a rather atypical characteristic for most DBMSs. Typical database administration tasks like installing and configuring the database server software, creating and optimizing tables, and creating views and stored procedures are simply not necessary with db4o. Adding a single small archive file (JAR or DLL) to your classpath gives access to the db4o API, which has all the classes you need to store and retrieve objects. db4o is incredibly simple to use, and its small footprint means that it opens up the use of object databases to a whole range of embedded applications."

All this makes it more suitable for big workloads and complex objects. As we will later discuss in our comparison the native queries are very intuitively, as are the queries by example.

Furthermore, db4o is not meant to be a stand alone database management system, as Paterson et al., (2006) puts it: " A key strength is its ability to be seamlessly embedded into a .NET or Java application, using a data model that is the same as the application's object model and without the need for database administration. There is no need to map objects to tables, and complex object models are easily supported." This can both be a pro and a con, but is mostly a given fact that should be considered when finding the proper DBMS for your situation.

Since db4o is not a string-based, it is not very suitable for full text indexing. This basically just means you should not use it when you are searching through texts, but then again, why would you; far better alternatives are available for that. More pros and cons for the three different types of queries that db4o supports will be given later on when we explain and compare these types.

Comparison

We are explaining db4o query syntax and comparing it to equivalents in SQL. We also take a look at the performance of db4o by discussing the results from a paper by Saxena and Pratap (2013).

Query syntax

db4o provides three different querying systems which are Query by Example(QBE), SODA, and Native Queries. In this part we going to explain how to store, retrieve, update, delete an object from the database also how to join two objects by using each type of queries.
First we should create a class to keep our data.

It looks like this:

```java
package db4o;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
public class Person {

    private String _name;
    private int _age;
    public Person(){}

    public Person(String name, int age) {
      _name = name;
      _age = age;
    }

    public int getAge(){
        return _age;
    }

    public void setAge(int value){
        _age = value;
    }

    public String getName(){
        return _name;
    }

    public void setName(String value){
        _name = value;
    }

    public String toString(){
        return "[" + _name + ";"+ _age + "]";
    }
}
```

Opening the database

To access the db4o database or create a new one call Db4o.openFile() and send the path of the database as parameter to obtain a ObjectContainer instance. ObjectContainer simply represents the database. So, our code looks like following:

```java
ObjectContainer mydb = null;
mydb = Db4o.openFile("C:/myDB/myDB.txt");
```

This code will open the database if the path exists, if not the database will be opened automatically.

Storing an object

To store an object we shall call store() function. We should first create an object and pass this object as parameter to store() function. It looks like following:

```
Person pinar = new Person("Pinar",24);
Person anneMarie = new Person("AnneMarie", 23);
mydb.store(pinar);
mydb.store(anneMarie);
```

For the previous version of db4o it is possible to use set() function as well. The version of the database which is used here is 7.4 and JDK is 1.6.

Displaying the database/result

In our code we wrote a function called *listResult()* to see the database.

```
public static void listResult(ObjectSet result) {
    System.out.println(result.size());
    while (result.hasNext()) {
        System.out.println(result.next());
    }
}
```

This function is usable when the result you want to see is in ObjectSet type. In other word while the queries are written using by query by example. For displaying the result which typed by using Native queries, we have another function which is called *listResultNQ().* This function receiving parameter in List<> type. It looks like this:

```
public static void listResultNQ(List<Person> person) {
    while ((ObjectSet)person.hasNext()) {
        System.out.println((ObjectSet)person.next());
    }
}
```

Query by Example (QBE)

When using Query-By-Example, you create a prototypical object for db4o to use as an example of what you wish to retrieve. db4o will retrieve all objects of the given type that contain the same (non default) field values as the example. The results will be returned as

an ObjectSet instance. We will use a convenience method *listResult()* to display the contents of our result ObjectSet.

Advantages:
-It is easy to use and very suitable for beginners. Most of the time it is recommended to use Native Queries only after getting more familiar with db4o.

Disadvantages:
-Not useful for complex queries

SELECT statement with WHERE clause

db4o query using QBE:

```
//finds the objects whose name value is "Pinar" and returns it/them.
Person proto = new Person("Pinar", 0);
ObjectSet result = mydb.queryByExample(proto); //Query-by-Example
listResult(result);
```

[Pinar;24]

SQL query:

```
//Same query in SQL
SELECT *
FROM Person
WHERE name = "Pinar"
```

QBE is limited in its power. It can only provide an equivalent for queries with criteria that are matched exactly, and it isn't very flexible when it comes to compound criteria (Paterson et al., 2006).

UPDATE an object

db4o query using QBE:

```
ObjectSet result = mydb.queryByExample(new Person("AnneMarie",0)); //updating an object
Person found = (Person)result.next();
found.setAge(25);
mydb.store(found);
retrieveAllPersons(mydb);
```

SQL query:

```
UPDATE Person
SET age =25
WHERE name=”AnneMarie”
```

DELETE an object

db4o query using QBE:

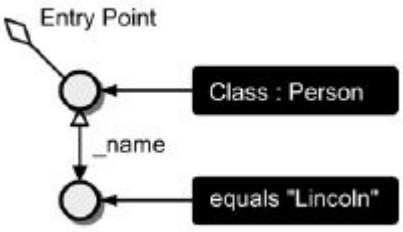
```
ObjectSet test = mydb.queryByExample(new Person("Esteban",0)); //deleting an object
Person found2 = (Person)test.next();
mydb.delete(found2);
```

SQL query:

```
DELETE FROM Person
WHERE name = "Esteban";
```

SODA Queries

SODA Queries were the first query method to be included in db4o, and internally, native queries still get translated to SODA queries. They are string-based graph queries, “where nodes represent classes or fields of classes, and edges represent relationships that can be traversed to reach nodes.” (Paterson et al., 2006)



In the image next to this you can see a SODA query graph representation from the book by Paterson et al. (2006) for retrieving all objects from the class Person that have an attribute _name that is equal to “Lincoln”. As you can see the field of the class ‘descends’ from the class node itself.

Advantages:

- Very fast
- Quite logical and efficient once you get the hang of it
- Good for dynamic query generation

Disadvantages:

- Harder to get used to, since it is such a different way of querying than SQL.
- String-based, so not type safe and not compile time checked

SELECT all

```
Query query = mydb.query();
query.constrain(Person.class);
ObjectSet res = query.execute();
listResultNQ(res);
```

[Gandhi;79]
[Pinar;24]
[AnneMarie;23]

SELECT with WHERE

```
Query query = mydb.query();
query.constrain(Person.class);
query.descend("_name").constrain("Gandhi"); // search a name
ObjectSet result = query.execute();
listResultNQ(result);
```

[Gandhi;79]

SELECT with WHERE NOT

```
Query query=mydb.query();
query.constrain(Person.class);
query.descend("_age").constrain(79).not(); // not 79
ObjectSet result = uery.execute();
listResultNQ(result);
```

[Pinar;24]
[AnneMarie;23]

SELECT with OR

```
Query query = mydb.query();
query.constrain(Person.class);
Constraint firstConstr = query.descend("_age").constrain(86); // first constraint
query.descend("_name").constrain("Pinar").or(firstConstr); // second, using or
ObjectSet result = query.execute();
listResultNQ(result);
```

[Pinar;24]

SELECT with AND

```
Query query = mydb.query();
query.constrain(Person.class);
Constraint firstConstr = query.descend("_age").constrain(24); // first constraint
query.descend("_name").constrain("Pinar").and(firstConstr); // second, using and
ObjectSet result = query.execute();
listResultNQ(result);
```

[Pinar;24]

Native Queries

NQs basically use the functionality the programming language gives you. It uses a method that returns a boolean value (true or false) depending on the result.

Advantages:

- Better for more complex queries in db4o.
- Subqueries are arguably easier to write and more readable in db4o than in SQL since the sequence of operations is more obvious (Paterson et al., 2006)
- No need to edit mappings or query strings when the model changes.
- Sorting is easier

Disadvantages:

- It is possible to miss some well-known SQL functions which you need to implement but probably be integrated into db4o with more effort than SQL. The functions could be aggregate ones like AVG, SUM, COUNT.

SELECT statement with WHERE clause

```
List<Person> person = mydb.query(new Predicate<Person>() {
    public boolean match(Person person1) {
        return person1.getAge() == 25;
    }
});
```

SELECT statement with range

```
List <Person> person=mydb.query(new Predicate <Person>() {
    public boolean match(Person person1) {
        return person1.getAge() < 24 || person1.getAge() > 60;
    }
});
```

[Gandhi;79]
[AnneMarie;23]

SELECT with AND

```
List<Person> persons = mydb.query(new Predicate<Person>() {
    public boolean match(Person person) {
        return person.getAge() > 24 && person.getName() == "Pinar";
    }
});
```

[Pinar;24]

SORTING

```
// Comparator
Comparator<Person> personComparator = new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
};

// Query
List<Person> result = mydb.query(new Predicate<Person>() {
    public boolean match(Person person) {
        return true;
    }
}, personComparator);
```

[AnneMarie;23]
[Gandhi;79]
[Pinar;24]

Multiple classes

Let's look at how to do make joins in db4o, for which we replicate the example in Paterson et al. (2006). To make join, we should have 2 different classes. First, we can create another class called "Pilot".

package db4o;

```
public class Pilot {
  private String name;
  private int points;
  public Pilot(String name,int points) {
    this.name=name;
    this.points=points;
  }
  public int getPoints() {
    return points;
  }
  public void addPoints(int points) {
    this.points+=points;
  }
  public String getName() {
    return name;
  }
  public String toString() {
    return name+"/"+points;
  }
}
```

And we can assign a vehicle to a Pilot. To do that, let's create another class called "Car".

```
public class Car {
  private String model;
  private Pilot pilot;

  public Car(String model) {
    this.model=model;
    this.pilot=null;
  }

  public Pilot getPilot() {
    return pilot;
  }

  public void setPilot(Pilot pilot) {
    this.pilot = pilot;
  }

  public String getModel() {
    return model;
  }

  public String toString() {
    return model+"["+pilot+"]";
  }
}
```

To store object:

```
Car car1 = new Car("Ferrari");
Pilot pilot1 = new Pilot("Michael Schumacher", 100);
car1.setPilot(pilot1);
db2.store(car1);
```

And our second object could be stored like this:

```
Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
db2.store(pilot2);
Car car2 = new Car("BMW");
car2.setPilot(pilot2);
```

To retrieve a car which is used by a specific Pilot, with other words to see a car by pilot name:

By QBE:

```
Pilot pilotproto = new Pilot("Rubens Barrichello", 0);
Car carproto = new Car(null);
carproto.setPilot(pilotproto);
ObjectSet result = db2.queryByExample(carproto);
listResult(result);
```

By using Native Queries:

```
final String pilotName = "Rubens Barrichello";
List<Car> results = db2.query(new Predicate<Car>() {
    public boolean match(Car car) {
        return car.getPilot().getName().equals(pilotName);
    }
});
listResultNQ_join(results);

// We have to modify the listResultNQ function like following
// since the class name has changed this time.
public static void listResultNQ_join(List <Car> car) {
    while (((ObjectSet) car).hasNext()) {
        System.out.println(((ObjectSet) car).next());
    }
}
```

By SODA Queries

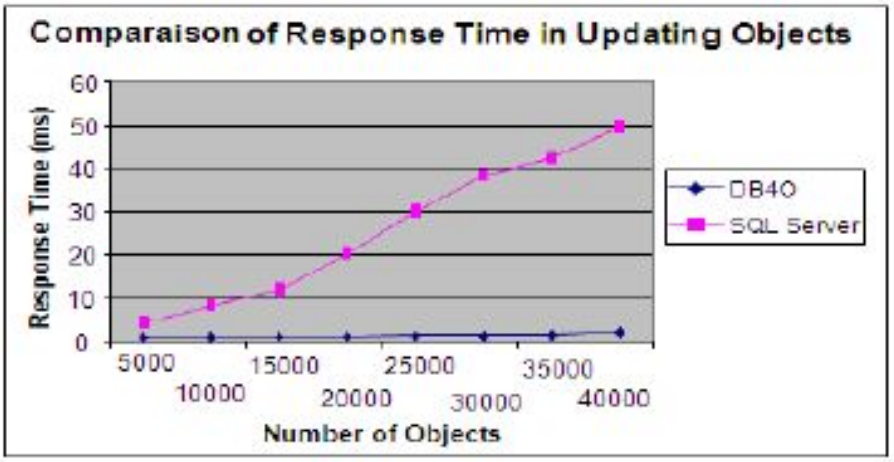
```
Query query = db2.query();
query.constrain(Car.class);
query.descend("pilot").descend("name").constrain("Rubens Barrichello");
ObjectSet result = query.execute();
listResult(result);
```

Query time / Performance

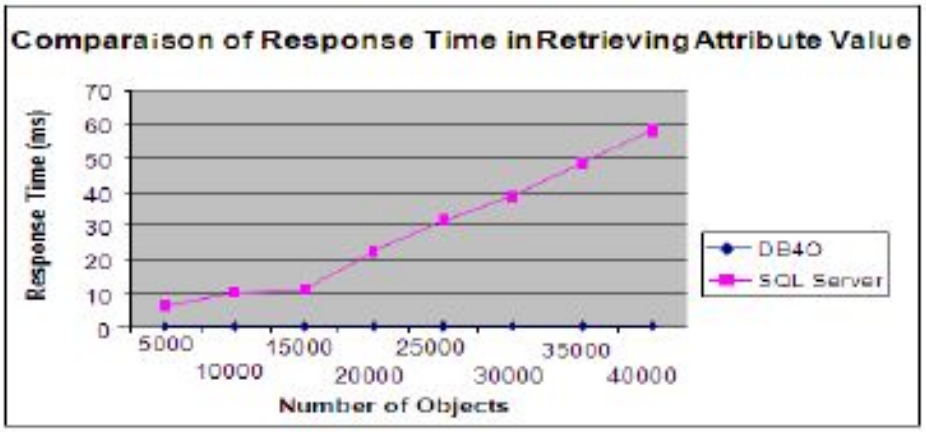
Vipin Saxena and Ajay Pratap wrote an article on ‘Performance Comparison between Relational and Object-Oriented Databases’ (2013) in which they compare the response times of db4o and SQL Server 2008 for writing, updating and retrieving objects. Since their research looks very well argumented, we decided not to replicate these results, but just report them. They build a database consisting of 40000 Users having fields UID, Name, Address and MobNumber. The field UID is the primary key and different objects of the User

class must have different IDs. The different customer objects were first inserted into the database, and then the objects were queried back by their IDs.

They found db4o took a bit more time when writing the objects, but was faster when updating or retrieving them, especially when dealing with a large database. This makes sense since objects do not have to be reassembled before updating or returning them. This gets further illustrated in the pictures below all from Saxena & Pratap (2013).

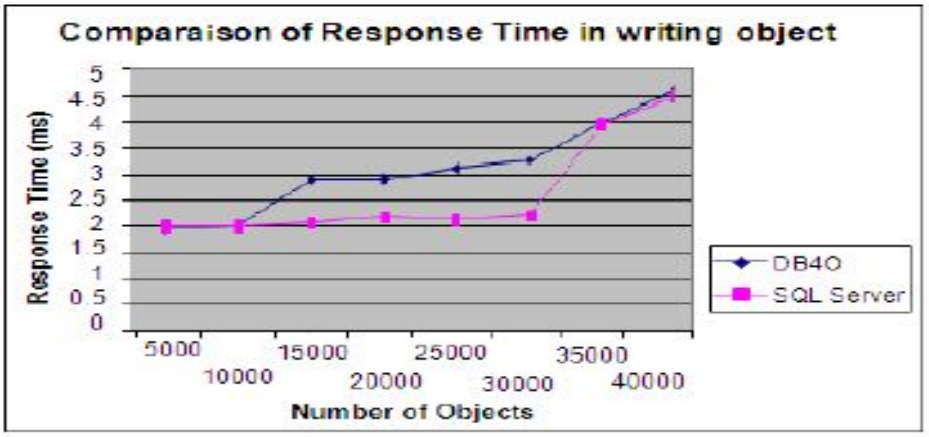


The above figure shows how SQL Server is performing compared to db4o when updating objects. As you can see is db4o always faster, but especially once the data set gets bigger. This makes sense because SQL Server needs to reassemble the object before being able to update it, and after the update it needs to disassemble it again. db4o however can directly access and alter the objects, which saves a lot of time.



The above figure compares the response time of DB4O and SQL Server depending on the number of data, when retrieving an attribute value. The response time gets higher for SQL

when the number of objects is increased. However, db4o stays virtually the same for all the period.



Writing objects is the only method tested by Saxena & Pratap (2013) on which db4o sometimes preformed worse than SQL Server. As gets illustrated by the picture above, is the writing time quite equal, just a millisecond slower in some of the tests they performed.

Conclusion

db4o was quite a breakthrough when it was released. It is part of the third-generation of databases and work significantly different than eg SQL, mostly by storing objects as they are and working natively. This is also the main reason it is way faster than SQL when updating or retrieving objects, especially on large databases. The native query and SODA query API's take a little bit of time getting used to, but are fast, smart and convenient. The native queries in particular are very intuitive and easy to work with. It is also very helpful that it works using Java or .NET, which prevents the impedance mismatch and is simply nice for programmers who prefer these languages over SQL.

However, db4o is not the proper choice for you if you want to do full text searches. Also, even though it is possible to link different classes, db4o is no relational database management system. Because of this, the system is still only used for quite specific applications.

On a more personal note: we were quite impressed by db4o. It is a very smart system, and the native queries are really intuitive and enjoyable if you like coding in Java. In regular databases you usually work with strings, which is not what this database system is designed for, but if you have a special case, and are looking for a DBMS to store objects, db4o is a great choice, especially when you're working with a big amount of complex objects.

Bibliography

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S. (1995). The Object-Oriented Database System Manifesto. [online] Available at: http://www.cs.cmu.edu/afs/cs.cmu.edu/user/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html [Accessed 4 Dec. 2018].

Bagui, S. (2003). Achievements and Weaknesses of Object-Oriented Databases. *The Journal of Object Technology, 2*(4), 29. doi:10.5381/jot.2003.2.4.c2. Available at: http://www.jot.fm/issues/issue_2003_07/column2.pdf [Accessed 16 Dec. 2018]

Hauser, P.(n.d.). Review of db4o from db4o objects. Available at: http://cis.bentley.edu/LWaguespack/CS630_Site/Downloads_files/OODBMS-db4o-Review.pdf [Accessed 14 Dec. 2018]

N.N. (n.d.) db4o-7.8-tutorial.[PDF File]. Retrieved from http://www-users.mat.umk.pl/~stencel/obd/db4o-7.8-tutorial.pdf [Accessed 14 Dec. 2018]

Odbms.org. I. (2018). Definition - ODBMS.org. [online] Available at: http://www.odbms.org/introduction-to-odbms/definition/ [Accessed 4 Dec. 2018].

Odbms.org. II. (2018). ODMG Standard - ODBMS.org. [online] Available at: http://www.odbms.org/odmg-standard/ [Accessed 4 Dec. 2018].

Paterson, J., Edlich, S., Hörning, H., & Hörning, R. (2006). *Definitive guide to db4o.* New York: Apress. Available at: https://www.researchgate.net/publication/229686866_The_Definitive_Guide_to_db4o [Accessed 14 Dec. 2018]

Saxena, V., & Pratap, A. (2012). Representation of Object-Oriented Database for the Development of Web Based Application Using Db4o. *Journal of Software Engineering and Applications*, 05(09), 687-694. doi: 10.4236/jsea.2012.59082. Available at: https://www.researchgate.net/publication/267940588_Representation_of_Object-Oriented_Database_for_the_Development_of_Web_Based_Application_Using_Db4o [Accessed 14 Dec. 2018]

Saxena, V., & Pratap, A. (2013). Performance Comparison between Relational and Object-Oriented Databases. *International Journal of Computer Applications*, 71(22). doi: 10.4236/jsea.2012.59082. Available at: https://pdfs.semanticscholar.org/bdeb/d4678cbbcd0d8d42a777e3921a9d5b7f531f.pdf [Accessed 14 Dec. 2018]