

# Bachelor Degree Project



## **PERFORMANCE ANALYSIS OF RELATIONAL DATABASES, OBJECT-ORIENTED DATABASES AND ORM FRAMEWORKS**

## **PRESTANDAANALYS AV RELATIONSDATABASES, OBJEKTORIENTERADE DATABASES OCH ORM-RAMVERK**

Bachelor Degree Project in Computer Science  
30 ECTS  
Spring term 2014

Victor Nagy

Supervisor: Henrik Gustavsson  
Examiner: Göran Falkman

## **Abstract**

In the planning stage of web and software development, it is important to select the right tool for the job. When selecting the database to use, relational databases like MySQL is a popular choice. However, relational databases suffer by object-relational impedance mismatch. In this work we will explore the response time of relational and object-oriented databases and the overhead of ORM frameworks. This will be done by creating a web application that receives data about flights and airports from a client, which measures the response time of the databases and the entire request. It was found that MySQL has the lowest response time, while the ORM framework Hibernate adds an overhead on some of the tests while performing similar to MySQL. Db4o had the highest response time in a majority of the tests. In future works, this study could be extended by other tests or by other type of databases.

**Keywords:** database, relational database, object-oriented database, ORM framework, web

# Table of contents

Table of contents.....	3
1 Introduction.....	1
2 Background.....	2
2.1 Object-oriented programming and persistent data in web applications.....	2
2.1.1 Java.....	2
2.2 Databases.....	2
2.2.1 Relational databases.....	2
2.2.2 Object-oriented databases.....	3
2.3 Object-Relational Mapping (ORM).....	4
2.3.1 Hibernate.....	7
3 Problem.....	8
3.1 Hypothesis.....	9
4 Method.....	10
4.1 Disadvantages with the method.....	10
4.2 Evaluation.....	11
4.3 Research ethics.....	11
5 Implementation.....	13
5.1 Literature study.....	13
5.2 Frameworks, tools and libraries.....	14
5.2.1 Play Framework.....	14
5.2.2 MySQL.....	15
5.2.3 Hibernate.....	17
5.2.4 db4o.....	20
5.3 Development.....	21
5.3.1 Server.....	21
5.3.2 Client.....	26
5.3.3 Client version 2.....	26
5.3.4 Conclusion.....	27
5.4 Pilot study.....	27
5.4.1 Relational - Inserting airports.....	27
5.4.2 Relational - Inserting flights.....	28
5.4.3 Relational - Searching flights.....	29
5.4.4 Relational - Listing all incoming flights to an airport.....	30
6 Evaluation.....	31
6.1 The Study.....	31
6.1.1 Summary.....	61
6.2 Analysis.....	62
6.3 Conclusion.....	64
7 Concluding Remarks.....	65
7.1 Summary.....	65
7.2 Discussion.....	65
7.3 Future work.....	66

# 1 Introduction

In many applications and websites there is a large need for persistence (Schahczenski, 2000). Persistence is the ability to save data between different executions of an application and between requests of a website. For example, a forum system needs to be able to save the posts the users make in case the information stored in memory is lost or the server is shut down (Kienzle and Romanovsky, 2002).

A way to ensure data persistency for an application is to use database software to save data (Liu et al., 2008). Large websites and applications that need to do a lot of data operations rely on the overall system to keep working even under high load. One bottleneck for the application is the database overhead incurred when inserting and retrieving data. This is defined as server-side latency (Ghosh and Rau-Chaplin, 2006). On one hand one might need short response times when it comes from a database, on the other hand, there is development time and effort that might lead you to pick a database that is easier to develop code for.

Relational databases are currently very popular (Bazghandi, 2006) but they are affected by the object-relational impedance mismatch (van Zyl et al., 2006). To solve that issue one could use an object-relational mapping (ORM) system or an object-oriented database. However, it might affect performance depending on the task.

In this work we will compare the MySQL relational database to the db4o object-oriented database and the Hibernate ORM framework to evaluate the time taken to do inserts to the database and how long it takes to search the database.

To do the comparison between the databases, a web application was developed. The web application uses the Play Java framework as a web framework and web server. The web application implements methods to insert and retrieve data from the MySQL relational database, from the db4o object database and to insert and retrieve data using the Hibernate ORM framework. A client application has also been developed to send requests to the server in a controlled manner.

## **2 Background**

### **2.1 Object-oriented programming and persistent data in web applications**

In the data model for object-oriented programming, real world entities are modeled as objects (Bagui, 2003). An object has a state and a behaviour. The state of the object represents the data stored by the object's attributes. The attributes could hold simple values like integers or they could hold values like arrays or other objects (Bagui, 2003).

Many applications require the use of persistent data (Schahczenski, 2000). The data—or state of the application—should be saved to the file system or a database. This data should still exist after the application has been terminated (Kienzle and Romanovsky, 2002). HTTP in itself is stateless and no information about the state of the connection is saved after the connection has been terminated (Goschka and Riedling, 1997) unless the application takes action to save it.

There are many available solutions for implementing persistence in applications. One tool for data persistence in Java is the Java Persistence API (JPA). Ohara et al. (2009) implements an e-commerce application that uses JPA as the persistence layer.

#### **2.1.1 Java**

Java is an object-oriented and platform independent programming language with features such as portability, flexibility and reusability (Prajapati and Dabhi, 2009).

When creating web applications with Java one could make use of Java Server Pages (JSP). In a paper by Prajapati and Dabhi (2009) they use JSP together with Enterprise Java Beans (EJB) to create a persistent web application.

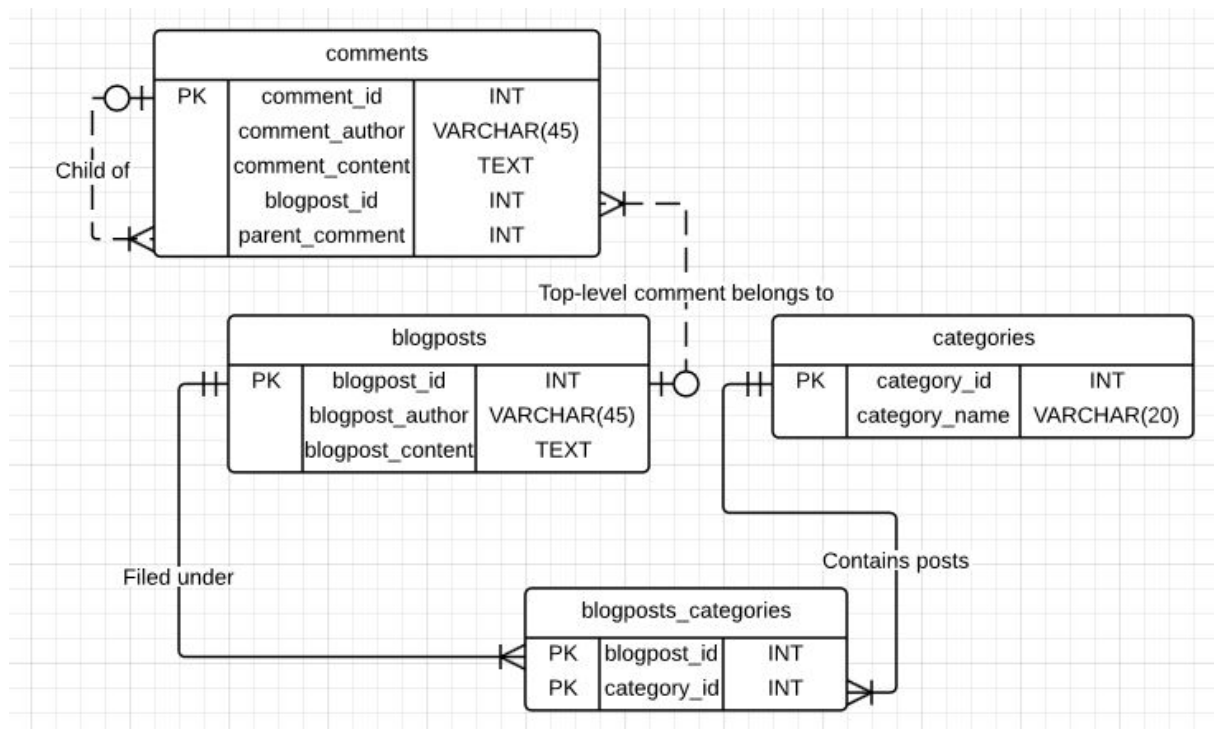
### **2.2 Databases**

The use of a database is one way to ensure data persistency in an application (Liu et al., 2008) but they put more load on the server, especially in an web environment where you have the web server, the application server and then the database server, all trying to interact with each other (Ghosh and Rau-Chaplin, 2006).

#### **2.2.1 Relational databases**

Relational Database Management Systems (RDBMSs) are very good for the generation of reports and data mining and databases can be used in systems like e-commerce or online social networks, but relational databases suffer from object-relational impedance mismatch (van Zyl et al., 2006) when used in object-oriented programming (OOP). OOP is built upon various concepts like objects to structure data, classes that define objects (in class-based object-oriented languages), inheritance, polymorphism, encapsulation and accessibility, none are fully supported by RDBMSs (Ireland et al., 2009). Relational databases are useful when querying flat data but it cannot directly query nested structures without rewriting the database (Mann and Devgan, 2000).

In a study by Mann and Devgan (2000) on storing large multimedia objects in a large video application built in PHP, they came to the conclusion that relational databases had faster querying time while object-relational databases had faster insertion.



**Figure 1** EER diagram of blog system

Figure 1 shows a relational database model of a simple blog system, modeled using an EER diagram. To map inheritances, the concrete table method has been used.

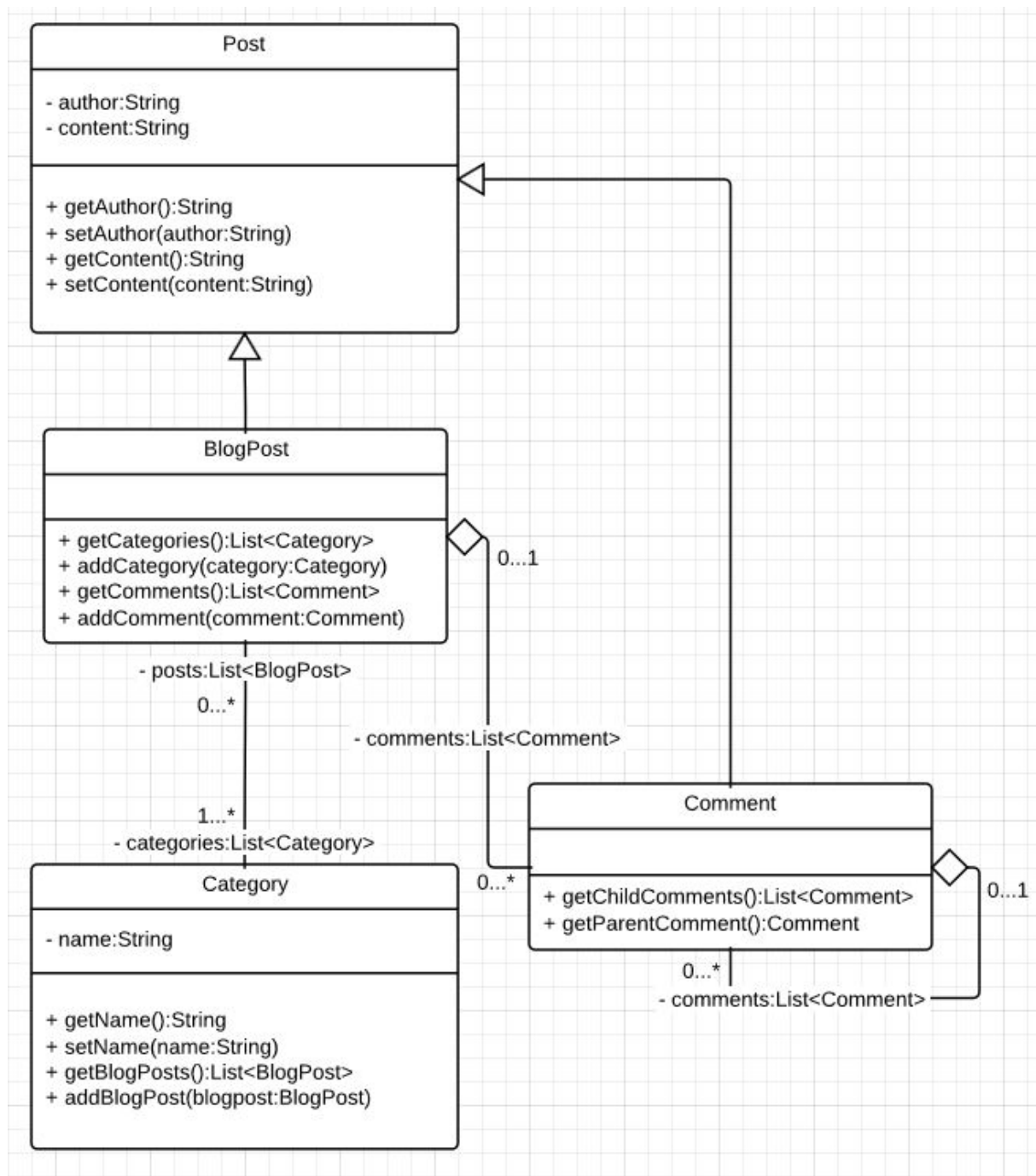
MySQL is a popular, open-source RDBMS that uses the Structured Query Language (SQL) to retrieve data from the database (Di Giacomo, 2005). It is common to use MySQL in both web applications and embedded applications.

## 2.2.2 Object-oriented databases

The Object-Oriented Database Management System (OODBMS) can lead to efficient implementation of object-oriented applications that depend on having persistent data (Lange, 1994). One advantage of using an OODBMS is that the database can store all the information related to an object (Lange, 1994). Another advantage of an OODBMS is that it is possible to easily navigate between objects in the object model, but at the same time it performs worse for sequential processing (van Zyl et al., 2006) which would be a disadvantage in an application that provides report generation as an example.

OODBMSs allow the user to define abstract data types. They allow the developer to add attributes and methods to a class and define inheritance, then allow for objects of that class to be stored in the database (Bagui, 2003). OODBMSs have built-in support for structures such as arrays and sets, unlike relational databases, which only support flat values (Schahczenski, 2000).

Since many languages used on the web are based on the object-oriented paradigm, object databases could possibly ease development. It is important to understand the needs of the application before choosing the database.



**Figure 2** An UML class diagram of a simple blog system

Figure 2 depicts the same database model as Figure 1, except it is the blog system's object-oriented representation, showing inheritance and associations.

DB4o is an object-oriented database system that can store Plain Old Java Objects (POJOs) (van Zyl et al., 2006). DB4o could be used either as an embedded database or as a standalone server. Unlike relational databases the developer does not need to define a separate data model for the application. The data model is defined by the application's class structure, where db4o can save and retrieve objects by looking at the class definition.

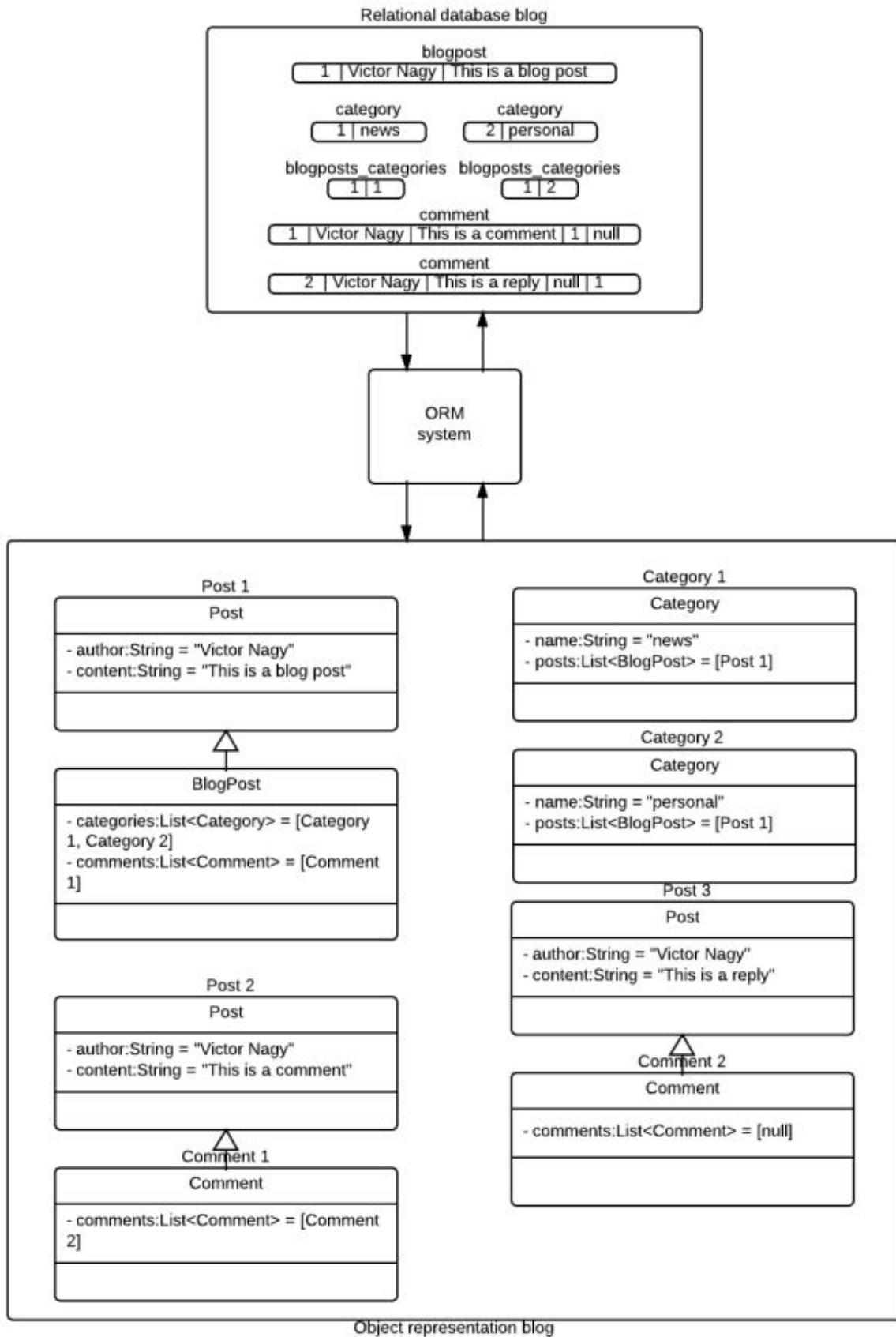
## 2.3 Object-Relational Mapping (ORM)

ORM is a way of addressing the problems of the object-relational impedance mismatch in relational databases (Ireland et al., 2009). The term ORM can be defined in different ways. For some, it is

used to refer to the transformation process from the relational paradigm to the object-oriented paradigm itself, while for others it is something defined in the configuration of ORM systems such as Hibernate (Ireland et al., 2009).

There are several disadvantages with ORM frameworks. One disadvantage is the need to define both the relational schema and the mapping files required for the object-relational mapping process (Walek et al., 2012). The developer has to manually create these mapping files based on the data types of an object on one side and the data types of the tables on the other side. These factors could lead to future problems in case you forget to update the mapping files after changing anything in the relational model. ORM frameworks could also lead considerable overhead in response time as mentioned in van Zyl et al. (2006) where they compared Hibernate and db4o.





**Figure 3** An ORM mapping between a relational and its object representation

*Figure 3* depicts how an ORM may map rows from the relational model (top) to the object-oriented model (bottom) and vice versa.

### **2.3.1 Hibernate**

Hibernate is an ORM library for the Java language. Hibernate stores objects and retrieves objects from a relational database (van Zyl et al., 2006). Hibernate supports any relational database as its database backend. Hibernate is an implementation of the Java Persistence API, which is the official standard for object-relational mapping in Java (Wegrzynowicz, 2013).

Hibernate maps properties from an object to a relational database and vice versa (van Zyl et al., 2006). It can also map inheritance and relationships. Hibernate uses either XML documents to define the mappings between the database and the objects, or it can use Java annotations defined in the Java code.

There are some disadvantages with Hibernate. It adds significant overhead when communicating with the database due to inefficient SQL queries being generated, too many SQL statements being executed and too many objects being loaded into memory (Wegrzynowicz, 2013)(van Zyl et al., 2006). Hibernate also require the developer to define the mapping manually, either in XML documents or in annotations (Walek et al., 2012)(van Zyl et al., 2006).

### 3 Problem

Different types of databases are optimal for different kinds of applications and while they can be made to conform to other use cases, they might not perform the same way. For example, relational databases are good for data mining and generation of reports (van Zyl et al., 2006). Relational databases are easy to use, but it can be difficult to store complex objects due to the object-relational impedance mismatch. Even so, relational databases are commonly used due to their widespread availability.

Using relational databases in an object-oriented application can be problematic due to the object-relational impedance mismatch, since you have to convert between the relational structured data and objects. To solve that, one could use ORM (van Zyl et al., 2006) tools and Active Record to ease development. Using those techniques adds (in some cases considerable) performance overheads as translation needs to occur, but it allows developers to use more powerful object-oriented modelling techniques (van Zyl et al., 2006). A correct decision on what to use is context based and requires a good understanding of the underlying issues (van Zyl et al., 2006).

When comparing relational and object databases, object databases are good for navigating around objects while relational databases are good for sequential processing and complex queries (van Zyl et al., 2006). In a study by Roopak et al. (2013) object databases insert data more efficiently and perform better than relational, while relational databases outperformed object databases when querying data.

While each database system might have different effects on development time and effort, many applications depend on the database system not being a bottleneck. This is especially true in web applications under high load.

In the study by Roopak et al. (2013) they compare the relational database MySQL against the object database db4o. Their conclusion was that the object database performed insert operations faster and more efficiently than the relational database, while the relational database performed select and delete operations faster than the object database. However, they did not test ORM tools at all.

Roopak et al. (2013) did not test relationships when it came to the relational database and object-oriented features on the object database, which could be seen as one of the core features of each database. This could be interesting to investigate as it could be useful to be aware of when making the decision of database to use.

van Zyl et al. (2006) compared the object database db4o against the ORM tool Hibernate with the help of the OO7 object database benchmark. Hibernate was found to be consistently slower than db4o.

Neither Roopak et al. (2013) or van Zyl et al. (2006) performed the evaluation in a web environment. Roopak et al. (2013) performed their evaluation offline directly against the database by inserting and retrieving flight data from the database. van Zyl et al. (2006) used the OO7 benchmark directly against the database.

A web server handles load differently than a database server. While a database server may be able to perform a thousand insert operations in a couple of seconds (Roopak et al., 2013), a web server might have trouble keeping up to the requests. Visitors to a website may give up visiting the site if it loads too slowly. It is important to make sure everything is as fast as possible and in this work we will focus on the database aspect.

*How much does the query execution time differ between relational databases, object-oriented databases and how much overhead does an ORM tool add to the relational database in a web environment? Is the difference small enough for choosing object-oriented databases or using ORM tools instead of pure relational databases if it eases development of a web application?*

### **3.1 Hypothesis**

We can conclude from the results found by Roopak et al. (2013) that db4o inserts data faster and more efficiently than MySQL, while MySQL retrieves data faster. We do not know the time taken for an ORM tool but our hypothesis is that the ORM adds considerable overhead to both inserting data and retrieving data, since there is an added layer between the application and the database.

## 4 Method

Studies regarding performance comparisons between object databases and ORM tools have previously been made by van Zyl et al. (2006). Roopak et al. (2013) compared relational databases and object databases. van Zyl et al. (2006) made a performance comparison of the db4o object-oriented database and the Hibernate ORM tool using the OO7 database test. Roopak et al. (2013) evaluated the db4o object-oriented database and the MySQL relational database using a database model of airline data.

In this paper we will build upon the study done by Roopak et al. (2013) by replicating their study while also comparing the results to an ORM tool. We will use the same airline database as used by Roopak et al. (2013), and do the same tests on both a relational database and an object database as well as the ORM tool. We will also add tests regarding relationships in the relational database by adding further tables of data and test object-oriented features of the object database.

The method that will be used is a quantitative method to investigate the performance differences between the two types of databases and the ORM tool by doing experiments. An advantage of the quantitative research method is that the quantitative data is a good way to do comparisons and statistical analysis (Wohlin et al., 2012). The experiments made in Roopak et al. (2013) will be repeated for the relational database MySQL and for the object database db4o while expanding on the evaluation by performing the same experiments on an ORM tool, in this case Hibernate.

We will also focus on a web environment to further extend Roopak et al. (2013). The evaluation will be performed in a web environment by implementing the test application as a web application in Java. Then a client will be developed to perform requests against the web application which will send data that will be inserted into the databases as well as retrieving data from the databases.

An alternative to experiments is to perform a case study. When performing a case study one could select the variables that represent the typical situation (Wohlin et al., 2012). That way a case study would provide a more realistic result. While a case study might be easier to plan the results generated are difficult to generalise and harder to understand, and one would have to do more analysis on the results to generalise it to other situations.

Case studies are good in a live environment and real context since they provide deeper understanding (Wohlin et al., 2012). They are aimed at investigating phenomena in their context while coping with the complex characteristics of real world phenomena. They are also more flexible than experiments as they don't require a strict bond between the test object and the environment.

With experiments you have more control over the test subjects, tests and environment than with case studies. Experimentation also allows opportunities for replication (Wohlin et al., 2012). With experimentation one could test whether a hypothesis or theory is true or not. Since our hypothesis is that ORM adds considerable overhead we can test whether it does or does not by performing experiments.

### 4.1 Disadvantages with the method

It is hard to simulate real use of an application using scripts to perform experiments. It is impossible to know exactly how users behave, in which order the user would perform an operation and how many times the user performs the various operations. This might lead to getting results that are true for the tests performed, but, not as true for the typical situation. Taking a large site with a lot of traffic for example, the load will most likely come in bursts and not spread evenly. Various operations and queries would be mixed as visitors load the site. In the tests performed in the experiment, the same operations would be repeatedly performed at even intervals.

We need to be certain that the experiments are performed exactly the same on all databases and the ORM systems to be able to compare the results. If different amounts of operations and different

types of operations would be made on each database you would not be able to compare them easily since different experiments were performed.

If you would go about using real users to simulate real usage you would run into the previously mentioned issues in addition to complications with the analysis of the results. To make sure the same tests are run on each database, the tests will be made in such a way that the tests are executed in exactly the same way and performs the exact same operations each time they are run.

Another point is the cost of making the experiment. Experiments can lead to high costs in both resources, time and money (Wohlin et al., 2012). Large-scale experiment might need special hardware to run, but that is not the case in this work as the hardware is already available. Some of the evaluations done during the experiment has the possibility to take very long time. Such issues would be a problem when renting hardware or having short and strict deadlines.

## 4.2 Evaluation

A database model will be created based on the airline data provided by American Statistical Association (ASA) (American Statistical Association, 2009a). This database model will be implemented on both the relational database and an object database then mapped to the ORM tool using the relational database as backend for the ORM tool. Supplemental data containing airports and carriers will also be used to create relationships between tables (also provided by the ASA (American Statistical Association, 2009b)).

As done by Roopak et al. (2013), we will measure the time it takes for the databases to insert data and the time it takes to retrieve data from the database. We will then measure how much overhead the ORM system adds on top of the relational database, the time taken to retrieve specific rows, and the time taken to query relationships, all of which were left out by Roopak et al. (2013). We will also measure time taken for the entire request, time spent before the database operation, and time spent after the database operation. These times will be held separate from each other to be able to see which parts take the longest time and which parts that change the most.

Roopak et al. (2013) compares the query execution time of MySQL and db4o. This is done by inserting different amounts of data from the flight data (1 000, 20 000, 40 000 and 80 000 pieces of data) into each database and recording the time taken for the operations to finish. Later on, select operations are executed on the data previously inserted to measure the time taken for each database to retrieve results. We will run the same tests on the same databases, but we will extend the study by adding tests regarding relationships and doing all tests against Hibernate, a free and open source ORM system developed in Java.

The requests will be executed in such a way that the web server won't overload completely. Performing a large amount of requests within a short period could easily overload the server (depending on the hardware). The server will initially be tested to see how many request it can handle without being overloaded (and possibly crashing) before doing the real tests.

## 4.3 Research ethics

The evaluation will be done without any test subjects. The data provided by the ASA does not contain any details that could identify a person as it only contains information about flights. We do not believe there are any ethical issues or cultural needs since no people are involved in the evaluation.

The databases were chosen based on popularity and availability and we do not have any association with the development of these softwares. The same is true for the ORM system. Both databases and the ORM system are open source and released for free under open source licenses.

The database models and all code will be released and attached to this work to ensure the possibility of repeating the evaluation. In that way, everything required to run the evaluation will be available. Due to the huge size of the airline data, this data won't be attached to the work and

anyone considering repeating the evaluation will have to download the data from the ASA website referenced in this work.

To allow verification of the results, hardware specifications as well as operating system and software versions will be included in the work.

The benchmarks will be developed as simply as possible without any unnecessary functionality to keep the benchmark as focused as possible. Prior to running the tests, all running programs and processes except for those required for running the operating system and the tests will be terminated as to not interfere with the testing.

## 5 Implementation

### 5.1 Literature study

For the implementation, some software and tools had to be selected. The main focus have been on the database software and the ORM tool. Some of the criterias we had when selecting these was that they should be free, open-source, easy to use and popular enough to ensure that if we get stuck at some point there would be help to get, either by documentation, tutorials, posts on various developer forums like StackOverflow or support channels on IRC (Internet Relay Chat).

MySQL was chosen as part of the replication of Roopak et al. (2013) as it is a widely used and popular RDBMS (Di Giacomo, 2005). MySQL is dual-licensed with an open-source (GPL) license and a proprietary license. In this work the open-source version was chosen since our goals are to use free and open-source software. MySQL covers all our criterias; it is easy to use, free and open-source, and the popularity makes it extremely easy to get help quickly if needed. MySQL's features cover our database model's needs very well and we see no obstacles at all since our database model is extremely simple. Other relational databases like SQLite and H2 were considered as well, but MySQL was deemed to be the best choice as it is used in the study by Roopak et al. (2013).

MySQL supports some more advanced features like replication, stored procedures, views and triggers. These, however, will not be used in our tests since they are not within the scope of what we will be testing.

Hibernate was chosen since it is one of the most popular ORM tools in Java. It has previously been compared to db4o in van Zyl et al. (2006), and its performance and optimisations were tested in van Zyl et al. (2009). Hibernate is both free and open source and is popular within the Java community. The basic features of Hibernate are easy to learn and use and it does not require much effort for a developer who has prior knowledge of using relational databases to learn how to use it.

A pitfall with Hibernate is that there are a lot of optimisation patterns that require full understanding of what you are doing and how it works. Wegrzynowicz, P. (2013) lists a couple of antipatterns regarding Hibernate that could affect performance negatively. A beginner who follows a tutorial or the getting started guide might chose the easiest or most obvious way to do the mappings, while it is not neccesarily the most optimal way to do it. These things could lead to issues in large applications where a lot of classes and tables have to be mapped, and where there is a constant load of visitors visiting the site in a web application, but since our application consists of two tables and two classes, we do not see this being an issue.

Hibernate covers our needs neatly since we do not need many advanced features to map the two tables in our database model. While the many advanced features of Hibernate may be useful in regular applications, they are not required for this work. Hibernate is also widely discussed on the Internet and it is easy to get help both on various developer forums and in the official IRC channel.

Db4o was chosen to cover the need of an object database. Db4o was compared to MySQL in the study by Roopak et al. (2013) that we are replicating. It was also used in the comparision of Hibernate against db4o by van Zyl et al. (2006). Db4o is fairly simple and straightforward to use to store objects. Db4o is both free and open-source, covering our criterias on that area. However, the latest version was released in 2011 (db4o, n.da) and doesn't seem to be in active development anymore. The db4o community is very limited and there are not many places to turn if something goes wrong. The documentation and tutorial released with db4o cover many topics briefly enough to start using db4o but are lacking in some areas (important notes to remember, troubleshooting etc).

Other than the disadvantages of not being very popular, it is easy to get started and fits our needs pretty well since our data model is very basic.



Last but not least, Play Framework was chosen as a web application framework. Play has a very clear way to create web applications and is extremely easy to learn, provided you already have knowledge in Java or Scala. Play is both free and open source (Play Framework, n.da) and fits all of our criterias of chosing tools for the work. Furthermore, it's easy to set up and development is fast. The documentation is a bit lacking and confusing to read since it is split into two parts (Java and Scala) with some topics being common, but play is popular enough that it is easy to find solutions to problems that can arise on the web, or in the official IRC channel.

The aim of Play is to focus on developer productivity and being more about development instead of configuration while still being flexible (Patel, 2011). Play makes it easier to develop web applications without the Java Enterprise Edition (Java EE) environment.

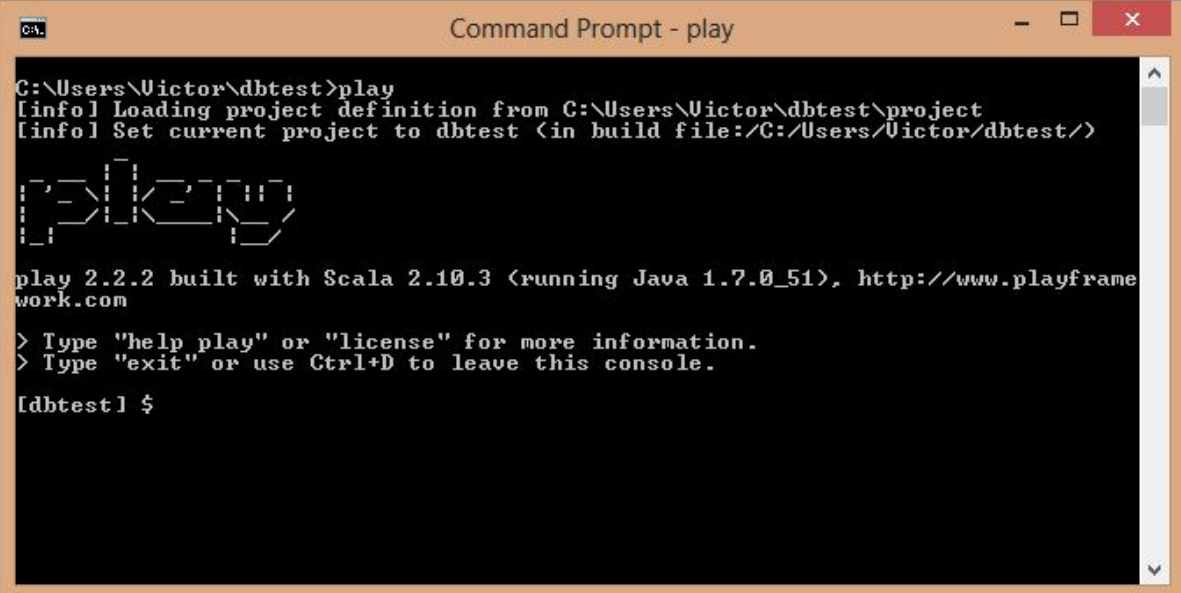
## 5.2 Frameworks, tools and libraries

### 5.2.1 Play Framework

To create the testing application, Play Framework will be used to ease development. Play Framework is an open source web application framework developed in Java and Scala (Play Framework, n.da). It contains a built-in web server, follows the Model-View-Controller (MVC) pattern, contains a templating engine and a routing engine, and lets you develop in both Java and Scala. It also supports hot code reloading, meaning you don't have to restart the server and/or application whenever you make changes to the code; changes are instantly visible at the next page reload (after a short wait while the changed files are compiled), which eases development.

Since the framework supports both Java and Scala code, a choice has to be made. We will choose Java since db4o has official support for Java. Some suggest db4o works in Scala since Scala runs on the Java Virtual Machine (JVM), some support are missing and incompatibilities exists and therefore Scala will not be used (Gao, 2012). The templating engine in the framework will not be used since it is out of the scope of the work, results will instead be outputted as plain text.

To create a play application, after downloading the framework and adding it to the system path, you have to open a console and execute the command **play new \*name of application\*** in the directory where you want the application to reside (Play Framework, n.db). To start the application and server, you have to enter the play console first by executing **play** followed by **run**.



```
C:\Users\Victor\dbtest>play
[info] Loading project definition from C:\Users\Victor\dbtest\project
[info] Set current project to dbtest (in build file:/C:/Users/Victor/dbtest/)

  _   _  _   _
  | | | | | | |
  |_|_|_|_|_|_|

play 2.2.2 built with Scala 2.10.3 (running Java 1.7.0_51), http://www.playframework.com

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[dbtest] $
```

Figure 4 The play console

The entry point of the application is the routing file where one defines the routes of the application (the paths in the URL a visitor would visit). In the routing file, located at **conf/routes** in the application

root, you would specify routes by HTTP method, path and which controller and method will handle the request.

File: \$projectroot/conf/routes

```
1 | # Home page
2 | GET      /                      controllers.Application.index()
3 |
4 | # Tasks
5 | GET      /tasks                 controllers.Application.tasks()
6 | POST     /tasks                 controllers.Application.newTask()
7 | POST     /tasks/:id/delete      controllers.Application.deleteTask(id:Long)
```

**Figure 5** Routing file from a todo application (Play Framework, n.db)

The paths can contain variables, for example the `:id` variable found on line 7 in *figure 5*. This serves as a placeholder and allows you to insert any value matching the datatype specified in the parameter of the controller method (in this case `Long`). More complex rules to control what values are allowed are also supported.

After each route a method that should handle the request needs to be specified. It is provided as the full path to the controller class and the method contained within.

File: \$projectroot/app/controllers/Application.java

```
1 | public class Application extends Controller {
2 |
3 |     public static Result index() {
4 |         return ok(index.render("Your new application is ready.")).
5 |     }
6 |
7 |     public static Result tasks() {
8 |         return TODO;
9 |     }
10 |
11 |     public static Result newTask() {
12 |         return TODO;
13 |     }
14 |
15 |     public static Result deleteTask(Long id) {
16 |         return TODO;
17 |     }
18 |
19 | }
```

**Figure 6** The Application controller from a todo application (Play Framework, n.db)

A controller contains static methods to handle the requests to the routes defined in the routing file. Here you would call upon your model to do the logic and return a **Result** object when everything is done. The controller is usually very simple and no logic should be implemented at this level except for request handling, passing data to the model and calling upon the view renderer to return a response to the user.

## 5.2.2 MySQL

MySQL is a table-based relational database management software that uses predefined database schemas to model the data (Quigley and Gargenta, 2006). The database schema defines the tables contained in the database, the columns in each table and relationships between columns and tables.

MySQL uses the Structured Query Language (SQL) to interact with the database, to create schemas and to insert, update and delete data in the database. SQL consists of two parts, the Data

Definition Language (DDL) and Data Manipulation Language (DML). DDL is used to modify data structures (create table, alter table, drop table, create index, drop index)(Quigley and Gargenta, 2006) and DML is used to manipulate data (selects, inserts, updates and deletes).

```
1 | CREATE TABLE posts (  
2 |   'id' INT AUTO_INCREMENT,  
3 |   'author' VARCHAR(32) NOT NULL,  
4 |   'content' TEXT NOT NULL,  
5 |   PRIMARY KEY 'id'  
6 | ) ENGINE=INNODB;  
7 |  
8 | CREATE TABLE comments (  
9 |   'id' INT AUTO_INCREMENT,  
10 |  'author' VARCHAR(32) NOT NULL,  
11 |  'content' TEXT NOT NULL,  
12 |  'post_id' INT NOT NULL,  
13 |  PRIMARY KEY 'id',  
14 |  FOREIGN KEY 'post_id'  
15 |    REFERENCES posts(id)  
16 |    ON DELETE CASCADE  
17 | ) ENGINE=INNODB;
```

**Figure 7** A database schema with 2 tables and a relation

In *figure 7* we define 2 tables (using DDL), **posts** and **comments**. They each define an **id** column with the **int** datatype. The **AUTO\_INCREMENT** keyword tells MySQL that this column should be automatically incremented for each row. Each table contain an **author** and **content** column. These columns are followed by the **NOT NULL** keyword to require the column to contain a value.

It is strongly recommended that each table contains a primary key. A column that contains a primary key cannot be null and is required to be unique (Effective MySQL, 2011). If there is no unique column in the table one could use an auto-incrementing integer. The primary key ensures that every row in the table can be uniquely identified as well as creating an index for faster searching. The value of the primary key is unique for the table, and two rows with the same value cannot exist. A primary key could consist of one or multiple columns. In case it consists of multiple columns, these columns together generate an unique key. In *figure 7* we define the **id** column as the primary key.

*Figure 7* defines a relationship between the **comments** and **posts**. This is done with the **foreign key** constraint. This puts a constraint between **post\_id** in the **comments** table and the **id** in the **posts** table. Only valid **id** values from the **posts** table are allowed to be inserted in the **post\_id** column and whenever a post with the **id** referenced in **post\_id** is deleted then all comments that reference that post are deleted as well.

```
1 | INSERT INTO posts (author, content) VALUES ("Nitori Kawashiro", "I love cucumbers");  
2 | INSERT INTO comments (author, content, post_id) VALUES ("Suwako Moriya", "Don't ask me  
what's inside my hat, okay?", 1);
```

**Figure 8** SQL statements to insert a row of data into the posts table and a row of data into the comments table

Inserting data into the database is straightforward, as can be seen in *figure 8*. To insert data, the table the data is to be inserted into is specified, followed by a list of columns (optional if you are going to insert data for every column in the correct order) followed by a list of values. It is important to ensure that the values being inserted passes the constraints of the columns, for example foreign key constraints, datatypes and lengths. Depending on the MySQL configuration, the datatype and length constraint is especially important to consider, otherwise you might end up with truncated and/or corrupted data.

```
1 | INSERT INTO `flights` VALUES ('1987', '10', '1', '4', '1134', '1133', '1246', '1237', 'TW', '59', 'NA', '192', '184', 'NA', '9', '1', 'STL', 'PHX', '1262', 'NA', 'NA', '0', 'NA', '0', 'NA', 'NA', 'NA', 'NA', 'NA');
```

**Figure 9** Insert of a single row of flight data (Roopak, 2013)

In *figure 9* all columns are specified in order and you do not need to specify the columns in this case as MySQL figures out which value goes where (however, MySQL will not figure out if you specified some of the values in the wrong order by mistake, unless they are the wrong type).

```
1 | SELECT * FROM posts WHERE author = "Cirno";
```

**Figure 10** A SQL query that searches the table **posts** for the author named **Cirno** and returns the entire row if the condition is met

**SELECT** queries are used to retrieve data from the database. Very basic data retrieval from a single table is very easy and straightforward, but the queries could be complex when multiple tables are joined together with complex conditions. You have to be careful with select queries when querying large amounts of data since badly designed queries can make a big hit on the time taken to retrieve a result. Returning more columns than needed may require more I/O-access and higher memory consumption, which could lead to performance hits (jOOQ, 2013). It is also important to take into account in which order to join tables, since the order can affect the execution time (Winand, 2012a). There are often several strategies to make complex queries, some more fit for some situations than others where it is a big hit on the performance instead.

Another thing to consider is to create indexes on the columns you are going to search in. Indexes create a search index in the database which makes select queries perform faster, since the database can look at the index to know where the requested data resides (as an example, if you have the range 0–10 and index each number into two groups—0–5 and 6–10—if you want to retrieve 3, you only need to look in the first groups since you can be certain it is not in the second group)(Winand, 2012b).

```
1 | SELECT * from flights;
```

**Figure 11** Query to retrieve all columns and all rows from the **flights** table (Roopak, 2013)

### 5.2.3 Hibernate

Hibernate is a free and open source ORM tool to make mapping between objects and relational databases easier. It makes it easier for developers to create persistent applications by using a relational database as data storage (Hibernate, n.da). The mapping between the relational and the object-oriented data model is done in mapping files or class annotations. It also lets you define the relationships between properties in different classes as they are defined in the database schema.

File: \$root/conf/hibernate.cfg.xml

```
1 | <?xml version="1.0" encoding="utf-8"?>
2 | <!DOCTYPE hibernate-configuration SYSTEM
3 |     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
4 |
5 | <hibernate-configuration>
6 |     <session-factory>
7 |         <property name="hibernate.dialect">
8 |             org.hibernate.dialect.MySQLDialect
9 |         </property>
10 |
11 |         <property name="hibernate.connection.datasource">DefaultDS</property>
12 |
13 |         <mapping resource="Airport.hbm.xml"/>
14 |         <mapping resource="Flight.hbm.xml"/>
15 |
16 |     </session-factory>
17 | </hibernate-configuration>
```

**Figure 12** The hibernate configuration file from the implementation of the testing software

To get started with Hibernate, a configuration XML file is needed. This file contains various settings: for example, which SQL dialect to use, connection settings, and mapping resources (Tutorialspoint, n.da). In *figure 12*, we use the JNDI datasource that the Play Framework provides, **DefaultDS**. We also specify that Hibernate should use the MySQL dialect and we add mapping resources to our Flight and Airport classes.

There are two alternatives to define mappings between objects and the database. One is to provide XML files that contains mapping definitions like properties, relationships and what field uniquely.

```
1 | <?xml version="1.0" encoding="utf-8"?>
2 | <!DOCTYPE hibernate-mapping PUBLIC
3 |     "-//Hibernate/Hibernate Mapping DTD//EN"
4 |     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5 |
6 | <hibernate-mapping>
7 |     <class name="Employee" table="EMPLOYEE">
8 |         <meta attribute="class-description">
9 |             This class contains the employee detail.
10 |        </meta>
11 |        <id name="id" type="int" column="id">
12 |            <generator class="native"/>
13 |        </id>
14 |        <property name="firstName" column="first_name" type="string"/>
15 |        <property name="lastName" column="last_name" type="string"/>
16 |        <property name="salary" column="salary" type="int"/>
17 |    </class>
18 | </hibernate-mapping>
```

**Figure 13** An Hibernate XML mapping file (Tutorialspoint, n.db)

The mapping files consist of several XML elements. The **class** element represents a class and contains each property to be mapped. The **name** attribute of the **class** element contains the canonical name (the full namespace and class, for example, **java.lang.String**), in *figure 13*'s case **Employee**.

Further, each **class** element is required to contain a **meta** element with a short description of the class and an **id** element that defines which column in the table is the unique id and how to generate it, however it does not need to be the primary key of the table (Hibernate, n.db).

The other way is to use annotations directly in the class. That way you do not need to separate the mappings from the logic. Annotations is a language feature in java where you add short keywords

before attributes, methods and the class itself. See **Appendix 1** for an example of an annotated class.

```
1 | <?xml version="1.0" encoding="utf-8"?>
2 | <!DOCTYPE hibernate-mapping PUBLIC
3 |   "-//Hibernate/Hibernate Mapping DTD//EN"
4 |   "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5 |
6 | <hibernate-mapping>
7 |   <class name="Employee" table="EMPLOYEE">
8 |     <meta attribute="class-description">
9 |       This class contains the employee detail.
10 |     </meta>
11 |     <id name="id" type="int" column="id">
12 |       <generator class="native"/>
13 |     </id>
14 |     <property name="firstName" column="first_name" type="string"/>
15 |     <property name="lastName" column="last_name" type="string"/>
16 |     <property name="salary" column="salary" type="int"/>
17 |     <many-to-one name="address" column="address"
18 |       class="Address" not-null="true"/>
19 |   </class>
20 |
21 | </hibernate-mapping>
```

**Figure 14** Example showing a many-to-one relation mapped in Hibernate (Tutorialspoint, n.dd)

Many-to-one relationship mapping is pretty straightforward. A many-to-one relationship is one of the most common associations where a row in a table can be associated with several other rows (Tutorialspoint, n.dd). In *figure 14* there is a many-to-one relationship defined between **Employee** and **Address**. The **name** attribute contains the name of the property that stores the **Address** object, while the **column** attribute specifies which column in the **EMPLOYEE** table contains the key and the **class** attribute specifies which class the row should be mapped to. **Appendix 2** contains the **Employee** class referenced in *figure 14*. In this example, an employee can only have one address (stored in the **address** property) while many employees can live on the same address.

```
1 | <?xml version="1.0" encoding="utf-8"?>
2 | <!DOCTYPE hibernate-mapping PUBLIC
3 |   "-//Hibernate/Hibernate Mapping DTD//EN"
4 |   "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5 |
6 | <hibernate-mapping>
7 |   <class name="Employee" table="EMPLOYEE">
8 |     <meta attribute="class-description">
9 |       This class contains the employee detail.
10 |     </meta>
11 |     <id name="id" type="int" column="id">
12 |       <generator class="native"/>
13 |     </id>
14 |     <set name="certificates" cascade="all">
15 |       <key column="employee_id"/>
16 |       <one-to-many class="Certificate"/>
17 |     </set>
18 |     <property name="firstName" column="first_name" type="string"/>
19 |     <property name="lastName" column="last_name" type="string"/>
20 |     <property name="salary" column="salary" type="int"/>
21 |   </class>
22 | </hibernate-mapping>
```

**Figure 15** Example showing a one-to-many relation mapped in Hibernate (Tutorialspoint, n.de).

See **Appendix 3** for the java class.

A one-to-many relationship could be seen as the inverse of a many-to-one relationship. Instead of

the **Employee** object having a reference to an **Address** object in *figure 14*, the **Address** object contains a list of **Employee** objects living at that address instead. In *figure 15* the **Employee** class contains a set to hold certificates of class **Certificate**. To implement this in the java class one would define a property with the **Set** datatype, in the case of *figure 15*, named **certificates**.

```
1 | String hql = "FROM Employee E WHERE E.id = 10";
2 | Query query = session.createQuery(hql);
3 | List results = query.list();
```

**Figure 16** Example of the Hibernate Query Language (Tutorialspoint, n.df)

Retrieving data using Hibernate is much like regular SQL. Hibernate Query Language (HQL) is very similar to regular SQL queries except you use the class name instead of the table name when defining where to look for the data.

### 5.2.4 db4o

db4o is an object-oriented database system that can store Plain Old Java Objects (POJOs) (van Zyl et al., 2006). db4o is released as an open-source project and developed by Versant, with the latest version released in 2011 (db4o, n.da).

```
1 | ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded
2 |     .newConfiguration(), DB4OFILENAME);
```

**Figure 17** Opening an embedded db4o database (db4o, n.db)

To open a db4o database when using db4o as an embedded database one will have to use the **Db4oEmbedded.openFile()** method which takes a configuration and a path to the database file and returns an **ObjectContainer**. If the database file does not exist, it is created automatically (db4o, n.db). When you are done with your database operations you should always close the database to release all resources associated with it.

```
1 | Pilot pilot1 = new Pilot("Michael Schumacher", 100);
2 | db.store(pilot1);
```

**Figure 18** Storing an object (db4o, n.db)

Storing an object is very simple, all you have to do is to make a call to the **store()** method, passing the object as an argument. After an object is stored, it can be retrieved, updated and deleted.

```
1 | Pilot proto = new Pilot(null, 0);
2 | ObjectSet result = db.queryByExample(proto);
```

**Figure 19** Retrieving an object by example (db4o, n.db)

There are several ways to retrieve objects from a db4o database. One way is to use **queryByExample()**. When using **queryByExample()** you construct a prototype object where you set the fields to what you are searching for. In *figure 18* an 'empty' prototype is created containing only default values, this will return a list of all **Pilot** objects stored in the database (db4o, n.db). **queryByExample()** has a limitation that it cannot search for objects if the search term is the default value of the datatype, since default values are disregarded (db4o, n.db). For example, trying to search for all pilots with a score of 0 will return all pilots since 0 is the default value of **int**. To overcome that limitation you either have to use one of the other ways to retrieve objects (native queries, SODA Query API) or do the filtering yourself.

```

1 | ObjectSet result = db
2 |     .queryByExample(new Pilot("Michael Schumacher", 0));
3 | Pilot found = (Pilot) result.next();
4 | found.addPoints(11);
5 | db.store(found);

```

**Figure 20** Updating an object in the database (db4o, n.db)

To update an object that is already stored in the database, you first have to retrieve it the same way as you usually retrieve objects. After the object has been retrieved you are free to modify it. To update the object in the database afterwards you call the **store()** method once more. The object being updated is required to either have been stored previously in the same session or retrieved from the database otherwise db4o will insert a new object in the database (db4o, n.db).

```

1 | Car car1 = new Car("Ferrari");
2 | Pilot pilot1 = new Pilot("Michael Schumacher", 100);
3 | car1.setPilot(pilot1);
4 | db.store(car1);

```

**Figure 21** Storing structured objects (db4o, n.db)

Sometimes an object may contain another object. Storing those are no issue. When calling **store()** on an object that contains another object, the contained object is stored automatically, and retrieved when the parent object is retrieved (db4o, n.db). It is also possible to store the objects separately. Updating a structured object is done the same way as a regular update (*figure 19*).

## 5.3 Development

### 5.3.1 Server

Software	Version
Db4o	8.0
MySQL Community Server	5.5.24-log
Hibernate	4.3.5-Final
Java	7
Play framework	2.2.2
Gson	2.2.4
MySQL connector	5.1.30

**Figure 22** the software used during development and the version of each software.

Setting up the development environment was easy enough. Having downloaded the framework and added it and the Java Development Kit (JDK) to the system path, a new Play project was created (refer to chapter 5.2.1).

The configuration of Play (**Appendix 5**) is left mostly with default values except for the database settings which were updated to connect to the local development MySQL database. A MySQL connection pool was also set up and the datasource bound to Java Naming and Directory Interface (JNDI) to let Hibernate take advantage of Play's connection pool instead of creating its own.

A MySQL connection pool of ten connections is provided by Play, all of these connections are



opened once you start the server. In initial development we would create a new connection for each request but it proved to be difficult as the connection was randomly lost when the server was busy handling requests. By recommendation from the official Play IRC channel the connection pool is used to solve that issue. A downside with that is that the time taken to open and close a database connection can not be measured in the tests.

A database connection can only do one query at a time and will block further operation until it is done (Stackoverflow, 2012). There is also a limit to the number of connections the database server will accept. A possible cause for the random dropping of the connection might be that connections were opened too fast for the database server to handle and the connection limit was hit before having time to close enough connections. A connection pool always has a defined amount of connections open. A thread will then request one of the available connections and when done, release it back into the pool. If there are no available connections, the pool can either open new ones or block until a connection becomes available, depending on the configuration of the pool.

```
File: $root/server/build.sbt
1 | name := "dbtest"
2 |
3 | version := "1.0-SNAPSHOT"
4 |
5 | libraryDependencies += Seq(
6 |   javaJdbc,
7 |   javaEbean,
8 |   cache,
9 |   "com.google.code.gson" % "gson" % "2.2.4",
10 |   "mysql" % "mysql-connector-java" % "5.1.18"
11 | )
12 |
13 | play.Project.playJavaSettings
```

**Figure 23** The build properties

Two maven libraries were added to the dependencies of the project. The first one is gson, which is a library for serializing objects to JavaScript Object Notation (JSON) and vice versa. This library is used both in the client and on the server to pass data between each other. The other library is the MySQL Java connector which is required to interact with MySQL from Java. Both of these dependencies are automatically downloaded when running the application for the first time.

Two versions of the application was developed. The first version was non-relational and contained only a single datatype and table. In this version every request would result in exactly one database operation. When the non-relational version was done the application was branched to a new version where the airports table was added and relationships were created between the flights and the airports. In the relational version, inserting flights is not a single operation anymore since the airports needs to be retrieved before being able to insert the flight.

```
File: $root/server/conf/routes
1 | POST    /insert/airport/one/:time    controllers.Insert.oneAirport(time: Long)
2 | POST    /insert/flight/one/:time  controllers.Insert.oneFlight(time: Long)
3 |
4 | GET      /select/flight/deptime/:depTime/:time
      controllers.Select.flightByDepTime(depTime: Int, time: Long)
5 | GET      /select/join/flight/dest/:dest/:time
      controllers.Select.joinFlightByDest(dest: String, time: Long)
```

**Figure 24** The routing file

Four routes were initially created. These routes define where the client should send the request to insert or retrieve data. The two **POST** routes handles the creation of airports and flights. In the

beginning we were planning to have two approaches at creating airports and flights. One approach was to create them one-by-one (hence the **one** in the route) and the other was to insert them in bulk. Unfortunately due to time constraints we were only able to implement the first approach (we will discuss these approaches more in-depth in the client chapter).

The two **GET** routes handle the requests to do the retrieval tests. The first one retrieves flights by the given departure time, and the second route retrieves all flights arriving at the given destination. No actual data is returned to the client, with a JSON array returned instead. The **:time** parameter in each route contains the timestamp when the client sent the request.

The first POST route and the last GET route is exclusive to the relational version of the application since the airports table does not exist in the non-relational version.

```
File: $root/server/app/models/database/Database.java
1 | package models.database;
2 |
3 | import models.datatypes.Airport;
4 | import models.datatypes.Flight;
5 |
6 | public interface Database {
7 |
8 |     public long insertAirport(Airport airport);
9 |     public long insertFlight(Flight flight);
10 |    public long selectFlightByDepTime(int depTime);
11 |    public long joinSelectFlightByDest(String dest);
12 | }
```

**Figure 25** The database interface

A database interface was created to define which methods each database and the ORM tool was required to contain. The interface contains two methods for inserting data into the database and two methods to retrieve data from the database. Interfaces are very useful when you want to provide several implementations that can be used interchangeably. The application does not need to be concerned about how the data is stored and retrieved. All the application needs is a way to store and retrieve data.

Two datatypes were used in the application, listed in **Appendix 8** and **Appendix 9**. These datatypes contain the same fields as the dataset provided by American Statistical Association (2009a). However, each datatype has an **id** property since it is a requirement of Hibernate that each object has a unique id (Hibernate, n.db). The Airport class also contain two lists with flights flying in and flying out from the airport.

When a request arrives at the server and Play decides which route to use, the method in the specified controller class is called. **Appendix 10** lists the **Insert** controller class which contains two methods, **oneFlight()** and **oneAirport()**. Each method is defined as static, as Play does not initialize objects of the class and has to be able to call the method without an object. A method in a controller is the first point we can execute our own code without making any modifications to the Play framework, therefore the first thing we do in each method is to take the current timestamp. We then create an array to hold our timestamps, which will later be returned to the client.

The Insert controller handles data sent as a **POST** request. To use the data sent with the request, Play's **Form** class can be used. The data from the client is an object serialized as JSON, so we use the gson library to deserialize the JSON to the corresponding objects.

The variable **db** holds the database. Thanks to the **Database** interface we can easily choose if we want to run the tests with MySQL, db4o or Hibernate and store each class in the same variable,

since each class is guaranteed to have the methods we need. In our application, the database is selected by removing the comment from the desired database and comment the others. Ideally, in a real application this would be handled in a configuration file but we decided our way was sufficient.

As we are working with timestamps both on the client side and server side, it is important to make sure that the clocks of the client and server computer are synced if you run the client and server on different computers, otherwise the timestamps will not be correct if the clocks differ. The reason for using multiple timestamps is to be able to measure different parts of the request.

**Appendix 11** lists the **select** controller class. It is similar to the Insert controller but we do not need to bother with fetching additional data and deserializing it since the **Select** calls are **GET** requests.

```
1 | PreparedStatement pst = conn.prepareStatement("INSERT INTO airports (iata, airport,
    city, state," +
2 | "country, lat, longitude) VALUES (?, ?, ?, ?, ?, ?, ?)");
3 | pst.setString(1, airport.getIata());
4 | pst.setString(2, airport.getAirport());
5 | pst.setString(3, airport.getCity());
6 | pst.setString(4, airport.getState());
7 | pst.setString(5, airport.getCountry());
8 | pst.setDouble(6, airport.getLat());
9 | pst.setDouble(7, airport.getLongitude());
```

**Figure 26** Prepared statements with parameters

**Appendix 12** lists the **MySQL** class, the MySQL implementation of the **Database** interface. When an object of the class is initialized a connection is retrieved from the connection pool by calling **DB.getConnection()**. Prepared statements with parameters are used for the database operations. Prepared statements together with the use of parameters (see *figure 26* for example of a prepared statement with parameters) avoids SQL injection by sending the parameters separately, not being part of the query itself. It also makes the code more readable and decreases the possibility of faulty string concatenation.

The **insertFlight()** method needs to fetch the **id** of the airports before being able to insert the flight itself. This is due to the client specifying the airport by its IATA code, while the database relation expects the **id** of the airport. This adds two **SELECT** queries to the request which affects the response time negatively compared to just doing a single insert.

The Hibernate implementation can be seen in **Appendix 13**. To be able to use Hibernate, a **SessionFactory** needs to be created. The **SessionFactory** could be seen as fulfilling the same functionality as the MySQL connection pool. Hibernate is configured to use the MySQL connection pool that Play provides through JNDI. The **SessionFactory** is first initialized by a call from the **Global** (see **Appendix 14**) class when the server starts and exists throughout the applications runtime.

Before inserting or retrieving objects, one has to create the mapping configuration. We chose to map the classes using separate XML mapping files (**Appendix 16** and **Appendix 17**) since we did not want to put annotations specific to an implementation directly in the classes, since they are not needed for the MySQL and db4o tests.

The mapping files are pretty straightforward (see *chapter 5.2.3* for further info on Hibernate and mappings). A unique auto-incrementing ID had to be added to the database tables and objects, as per the requirements of Hibernate. This column was not originally present in the dataset.

Two many-to-one relationships were added to the **Flight** mapping in **Appendix 16**. A flight can have a single destination and a single origin while an airport can have several flights flying in and several flights flying out from the airport.

In the **Airport** mapping in **Appendix 17** two one-to-many relationships were added. These are lists to all flights departing and arriving at the airport. They are defined as inverse relations, since it is **Flight** that owns the relationship (the **Airport** id is referenced from **Flight**). At first a set was used to map the relationship, but we later changed the approach to use a bag instead (corresponds to java collections and lists), since using a set in this situation might lead to poor performance (Wegrzynowicz, 2013).

When inserting objects using Hibernate, one has to retrieve a **Session** from the **SessionFactory**. The next step is to start a transaction and save the object. You have to make sure to commit and close the transaction when you are done to make sure that the connection is released to the pool again.

In the **insertFlight()** method, the same flow is required as in the MySQL implementation. The destination and origin airport needs to be retrieved and added to the flight, before the flight itself is saved to the database.

When doing the last test, **joinSelectFlightByDest()**, we had to retrieve the linked list stored in the **Airport** object and save its size since Hibernate does not retrieve data from relationships until they are called by default (lazy loading).

**Appendix 18** lists the db4o implementation of the **Database** interface. Similar to Hibernate, the database file of the embedded db4o database is opened during the startup of the server. Opening and using the database with default configuration is pretty easy since it can be done in a one-liner: **ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "db4o.db");**.

Storing an object is as easy as doing **db.store(airport)**. Unlike MySQL and Hibernate, the database does not need to be closed until you are done with the database (when you stop the server). Like MySQL and Hibernate, the destination and origin airport needs to be retrieved and the destination and origin of the flight from the request needs to be replaced by those retrieved from the database. The flight is added to the lists contained in the airport object to create the bi-directional reference between **Flight** and **Airport**.

Initially, we had a lot of issues with the **insertFlight()** db4o implementation. The lists containing arriving and departing flights would not save properly, either being left empty or have too many flights in them. We managed to fix it by changing the order that the objects were saved in. Initially, an **Airport** was retrieved, the **Flight** added to the correct list of the **Airport** object and the destination airport/origin airport of the flight was set with the new **Airport** object. Then the **Airport** was stored and after that the **Flight** was stored. This seemed to be an issue and we could not gather from the documentation why this was incorrect. After reordering how the objects were assigned and stored the issue went away, leading us to believe that the order of execution is important.

There were issues using db4o together with the Play framework. In the beginning, it was not possible to store objects since an exception (**models.datatypes.Flight cannot be cast to com.db4o.reflect.generic.generic object**) was thrown and the server crashed. A search on the Internet did not provide any helpful suggestions and we were left on our own to solve this issue. We did not know why but it seemed like our class was mistakenly passed as an argument at one point and db4o tried to typecast it to a different type. In the end we tried starting Play using Play's production environment instead of it's development environment and that solved the issue. Our theory is that since Play supports hot code replacement in the debug environment, it did not play well together with db4o which expected that the code of the application will not be replaced during runtime.

For the database schema used in the application, check **Appendix 22**.

### 5.3.2 Client

To be able to read the cvs files provided by American Statistical Association (2009a) and American Statistical Association (2009b) a Util class was created (**Appendix 19**). It has two methods, one that loads flights from the provided cvs and returns a list of flights and the other loads airports from the provided cvs and returns a list of airports. This is a time consuming task and therefore it is done before doing any requests.

As cvs are just text files where each line represents a row and each column is separated using commas, it is easy to parse this with a scanner. Objects are created and populated with the values from the scanner then added to a list which is then returned. Two help methods were required, **checkInt()** and **checkString()**, since the values in the provided cvs do not follow the correct datatypes (when a number is not available for a row, it is listed as 'NA'. The same applies to columns that are limited to one character in the database.). Since the cvs containing flights contain a bit more than one million flights, it is not feasible to store the data in memory. Therefore, there is a set limit of objects that will be created from the dataset.

A **HttpClient** class was created as an easy way to send **POST** and **GET** requests (see **Appendix 20**) based on code by Mkyong (2013). The **sendPOST()** method takes a URL and a string of JSON as arguments and makes a **POST** request against the URL. After a response has been received, a new timestamp is taken and added to the response. Some stats about the current test is taken, for example the total execution time (by subtracting the first index of the array from the last and added to a total) and number of requests. These are stored as static variables which are accessed from the main class. The **sendGET()** method is almost identical with **sendPOST()** except it sends a **GET** request instead, and takes no parameters.

The Main class (**Appendix 21**) is simple. It contains the four tests that will be made and you have to uncomment the test you want to execute. It loads either flights or airports to a list which is then iterated. In each iteration, a request relevant to the test is sent to the server and the response is printed together with useful statistics.

There are multiple strategies for sending requests. The one that is currently implemented sends one request with one database operation and when there is a response, it immediately sends another one. This will cause a continuous load on the server. Another strategy would be to send several requests at the same time, wait for them to execute and then send another several requests. A third would be similar to the second but as soon as one request is done, send a new one, keeping constant number of requests to the server. There are also two strategies on how to send the payload. The first strategy is to send a request that would perform the database operation on the server once, for example, a single airport. The second strategy is to send a request that would perform the database operation several times, for example if ten airports are sent per request. All these were considered but due to a lack of time, only one strategy was implemented. The strategy implemented is one database operation per request, and one request being sent as soon as the previous request is done.

### 5.3.3 Client version 2

In the previous client, all results were outputted into the system console. That is very inconvenient since most consoles have a limit of number of lines it would show, as well as the speed you can write to it. It would also be nearly impossible to automate a test suite that way, since old responses and results would disappear as new responses are written.

Therefore a new client was made, based on the old one. Instead of printing all of the responses to the system console, they are now saved to a MySQL database running on the client computer.

A simple connection pool was created to hold connection (**Appendix 23**). This was done since we did not want to slow down the sending of request by doing database operations on the same thread as the requests. The database operations were moved to a separate thread for each operation instead, where each thread would take a connection from the connection pool.

The HttpClient class was remade to return the array of timestamps instead of printing out responses and numbers to the system console (**Appendix 24**).

To be able to do tests more easily and in an automated way, a **Test** class was made (**Appendix 25**). The **Test** class would insert information about the test into the **tests** table and save the generated id.

Four tests were implemented (**Appendix 26**, **Appendix 27**, **Appendix 28** and **Appendix 29**). When one of the classes are initialised as an object, an id is sent to the superclass indicating which test that is being executed. A thread pool is created, which is used to execute the database operations. The data is processed just like before and sent to the server. When a response from the server is received, a new **Runnable** is created with the database operations and it is sent to the thread pool for execution. When all requests have been sent, the pool is shutdown and all threads that are still being executed are waited upon before continuing with the next test.

The Main class have been updated as well (**Appendix 30**). A couple of loops have been added controlling which dataset to use, how many repetitions and a loop for the databases. This makes the client fully automated without the need of supervision. A route has been added to the server to purge the databases after each database has been tested.

For the database used to store the result, see **Appendix 32**.

To calculate the results of each test, another small application was created to ease calculation and entry into a spreadsheet where graphs could be made (**Appendix 33**).

### 5.3.4 Conclusion

After developing the testing application, we are certain that we can measure our hypothesis and problem. By looking at the timestamps generated, we can see how much time each part of the request took to execute, and by looking at specific values in the timestamp array, we can make certain how long the database operation was.

## 5.4 Pilot study

The pilot study was performed with the first version of the client, both client and server running on the same computer

### 5.4.1 Relational - Inserting airports

```
...
[1397753966498,1397753966498,1397753966498,1397753966499,1397753966499] Request time: 1ms
5979ms | 5,979s | average (ms): 1,772 | requests made: 3374 | low: 1ms | high: 6ms
[1397753966500,1397753966500,1397753966500,1397753966501,1397753966501] Request time: 1ms
5980ms | 5,980s | average (ms): 1,772 | requests made: 3375 | low: 1ms | high: 6ms
[1397753966501,1397753966501,1397753966502,1397753966503,1397753966503] Request time: 2ms
5982ms | 5,982s | average (ms): 1,772 | requests made: 3376 | low: 1ms | high: 6ms
```

**Figure 27** Inserting 3 376 airports using pure MySQL

3 376 airports were loaded from the dataset and inserted into the database by sending requests to the server. In total, it took 5 982ms to insert all airports and get a result back from the server. That leaves the average time per request at 1.772ms, with a low at 1ms and high at 6ms.

```

...
[1397756758998,1397756758999,1397756758999,1397756759002,1397756759002] Request time: 4ms
15881ms | 15,881s | average (ms): 4,707 | requests made: 3374 | low: 2ms | high: 34ms
[1397756759002,1397756759003,1397756759004,1397756759006,1397756759006] Request time: 4ms
15885ms | 15,885s | average (ms): 4,707 | requests made: 3375 | low: 2ms | high: 34ms
[1397756759007,1397756759008,1397756759008,1397756759010,1397756759011] Request time: 4ms
15889ms | 15,889s | average (ms): 4,706 | requests made: 3376 | low: 2ms | high: 34ms

```

**Figure 28** Inserting 3 376 airports using Hibernate

Hibernate took a lot longer to complete the airport inserts than MySQL. The requests took in total 15 889ms to execute, which is 9 907ms longer than MySQL. On average a request took 4.706ms to complete.

```

...
[1397759126876,1397759126877,1397759126878,1397759126880,1397759126880] Request time: 4ms
15133ms | 15,133s | average (ms): 4,485 | requests made: 3374 | low: 0ms | high: 32ms
[1397759126880,1397759126881,1397759126882,1397759126884,1397759126884] Request time: 4ms
15137ms | 15,137s | average (ms): 4,485 | requests made: 3375 | low: 0ms | high: 32ms
[1397759126884,1397759126885,1397759126885,1397759126888,1397759126888] Request time: 4ms
15141ms | 15,141s | average (ms): 4,485 | requests made: 3376 | low: 0ms | high: 32ms

```

**Figure 29** Inserting 3 376 airports using db4o

db4o performed similar to Hibernate when inserting airports. The result is 15 141ms to execute all 3 376 requests with an average of 4.485ms per request. That makes db4o 748ms faster than Hibernate, resulting in Hibernate performing the worst amongst the 3 tools tested.

## 5.4.2 Relational - Inserting flights

```

...
[1397754179130,1397754179131,1397754179131,1397754179135,1397754179135] Request time: 5ms
15577ms | 15,577s | average (ms): 5,196 | requests made: 2998 | low: 4ms | high: 27ms
[1397754179135,1397754179135,1397754179136,1397754179140,1397754179140] Request time: 5ms
15582ms | 15,582s | average (ms): 5,196 | requests made: 2999 | low: 4ms | high: 27ms
[1397754179140,1397754179140,1397754179141,1397754179145,1397754179145] Request time: 5ms
15587ms | 15,587s | average (ms): 5,196 | requests made: 3000 | low: 4ms | high: 27ms

```

**Figure 30** Inserting 3 000 flights using pure MySQL

After 3 000 flights were inserted, the final time ended up at 15 587ms with an average of 5.196ms per request. Since inserting flights requires two extra selects to be executed to retrieve the id of the airport, each request take longer time than inserting airports.

```
...
[1397757048591,1397757048591,1397757048592,1397757048599,1397757048600] Request time: 9ms
31120ms | 31,120s | average (ms): 10,380 | requests made: 2998 | low: 8ms | high: 31ms
[1397757048600,1397757048600,1397757048600,1397757048608,1397757048609] Request time: 9ms
31129ms | 31,129s | average (ms): 10,380 | requests made: 2999 | low: 8ms | high: 31ms
[1397757048609,1397757048610,1397757048610,1397757048618,1397757048618] Request time: 9ms
31138ms | 31,138s | average (ms): 10,379 | requests made: 3000 | low: 8ms | high: 31ms
```

**Figure 31** Inserting 3 000 flights using Hibernate

Even when inserting the flights, Hibernate performed considerably worse than MySQL regarding response time. It took 31 138ms to insert the same 3 000 flights into the database, which is 15 551ms longer than pure MySQL. On average a request took 10.379ms to execute.

```
...
[1397759574098,1397759574099,1397759574100,1397759574133,1397759574134] Request time: 36ms
95016ms | 95,016s | average (ms): 31,693 | requests made: 2998 | low: 15ms | high: 49ms
[1397759574134,1397759574135,1397759574135,1397759574171,1397759574172] Request time: 38ms
95054ms | 95,054s | average (ms): 31,695 | requests made: 2999 | low: 15ms | high: 49ms
[1397759574172,1397759574173,1397759574174,1397759574207,1397759574207] Request time: 35ms
95089ms | 95,089s | average (ms): 31,696 | requests made: 3000 | low: 15ms | high: 49ms
```

**Figure 32** Inserting 3 000 flights using db4o

At this test, db4o performed worst of all tools. Taking 95 089ms to insert the 3 000 flights with an average of 31.696ms per request. Db4o was in total 79 502ms slower than MySQL at the same test.

### 5.4.3 Relational - Searching flights

```
...
[1397755021678,1397755021679,1397755021679,1397755021685,1397755021685] Request time: 7ms
23207ms | 23,207s | average (ms): 7,741 | requests made: 2998 | low: 6ms | high: 29ms
[1397755021685,1397755021686,1397755021686,1397755021692,1397755021693] Request time: 8ms
23215ms | 23,215s | average (ms): 7,741 | requests made: 2999 | low: 6ms | high: 29ms
[1397755021693,1397755021693,1397755021694,1397755021700,1397755021700] Request time: 7ms
23222ms | 23,222s | average (ms): 7,741 | requests made: 3000 | low: 6ms | high: 29ms
```

**Figure 33** Searching for 3 000 flights, one per request, using pure MySQL (3 000 flights in database)

Searching flights in the database took considerably longer time than inserting flights. The total execution of requests took 23 222ms with an average of 7.741ms per request.

```
...
[1397757434734,1397757434734,1397757434734,1397757434740,1397757434741] Request time: 7ms
25669ms | 25,669s | average (ms): 8,562 | requests made: 2998 | low: 6ms | high: 32ms
[1397757434741,1397757434742,1397757434742,1397757434748,1397757434749] Request time: 8ms
25677ms | 25,677s | average (ms): 8,562 | requests made: 2999 | low: 6ms | high: 32ms
[1397757434749,1397757434749,1397757434749,1397757434756,1397757434757] Request time: 8ms
25685ms | 25,685s | average (ms): 8,562 | requests made: 3000 | low: 6ms | high: 32ms
```

**Figure 34** Searching for 3 000 flights, one per request, using Hibernate (3 000 flights in database)

Hibernate did not perform much worse regarding searching for flights. The total execution result is 25 685ms which is only 2 463ms slower than MySQL.

```
...
[1397760274847,1397760274848,1397760274848,1397760274854,1397760274855] Request time: 8ms
24614ms | 24,614s | average (ms): 8,210 | requests made: 2998 | low: 6ms | high: 33ms
[1397760274855,1397760274856,1397760274856,1397760274862,1397760274863] Request time: 8ms
24622ms | 24,622s | average (ms): 8,210 | requests made: 2999 | low: 6ms | high: 33ms
[1397760274863,1397760274864,1397760274864,1397760274870,1397760274871] Request time: 8ms
24630ms | 24,630s | average (ms): 8,210 | requests made: 3000 | low: 6ms | high: 33ms
```

**Figure 35** Searching for 3 000 flights, one per request, using db4o (3 000 flights in database)



Even db4o performed in a similar way as MySQL and Hibernate. Making 3 000 searches took 24 630ms in total which is 1 408ms slower than MySQL.

#### 5.4.4 Relational - Listing all incoming flights to an airport

```
...
[1397756271200,1397756271200,1397756271201,1397756271208,1397756271209] Request time: 9ms
30007ms | 30,007s | average (ms): 10,009 | requests made: 2998 | low: 0ms | high: 23ms
[1397756271209,1397756271209,1397756271209,1397756271217,1397756271217] Request time: 8ms
30015ms | 30,015s | average (ms): 10,008 | requests made: 2999 | low: 0ms | high: 23ms
[1397756271217,1397756271218,1397756271218,1397756271225,1397756271226] Request time: 9ms
30024ms | 30,024s | average (ms): 10,008 | requests made: 3000 | low: 0ms | high: 23ms
```

**Figure 36** Listing incoming flights using pure MySQL (3 000 flights, 3 376 airports in database)

Searching for airports and joining it together with the flight table took in total 30 024ms to execute, with an average of 10.008ms per request. Joining tables adds time to execution depending on the size of the tables being joined together.

```
...
[1397757896419,1397757896420,1397757896420,1397757896433,1397757896434] Request time: 15ms
50809ms | 50,809s | average (ms): 16,948 | requests made: 2998 | low: 7ms | high: 43ms
[1397757896434,1397757896434,1397757896434,1397757896447,1397757896447] Request time: 13ms
50822ms | 50,822s | average (ms): 16,946 | requests made: 2999 | low: 7ms | high: 43ms
[1397757896447,1397757896448,1397757896448,1397757896461,1397757896462] Request time: 15ms
50837ms | 50,837s | average (ms): 16,946 | requests made: 3000 | low: 7ms | high: 43ms
```

**Figure 37** Listing incoming flights using Hibernate (3000 flights, 3376 airports in database)

Hibernate performed poorly on the listing test when compared to MySQL. Taking 50 837ms to execute with an average of 16.946ms, it took 20 813ms longer time to execute than MySQL.

```
...
[1397761497633,1397761497634,1397761497634,1397761497644,1397761497645] Request time: 12ms
37739ms | 37,739s | average (ms): 12,588 | requests made: 2998 | low: 0ms | high: 191ms
[1397761497645,1397761497645,1397761497645,1397761497656,1397761497657] Request time: 12ms
37751ms | 37,751s | average (ms): 12,588 | requests made: 2999 | low: 0ms | high: 191ms
[1397761497657,1397761497657,1397761497657,1397761497669,1397761497669] Request time: 12ms
37763ms | 37,763s | average (ms): 12,588 | requests made: 3000 | low: 0ms | high: 191ms
```

**Figure 38** Listing incoming flights using db4o (3000 flights, 3376 airports in database)

In the listing test, db4o performs a little bit slower than MySQL. The end result is 37 763ms to execute all requests with an average of 12.588ms per request, which puts db4o 7 739ms slower than MySQL in total execution time.

## 6 Evaluation

### 6.1 The Study

The tests will be performed with various amounts of flights: 1000, 10 000 and 20 000 flights. Due to relationships between flights and airports, all 3 376 airports will have to be inserted to make sure the relationship constraints are met when inserting flights.

Hardware	Type
RAM	Corsair XMS3 Vengeance DDR3 PC12800/1600MHz CL9 3x4GB (12GB)
CPU	Intel Core i7 950, 3.07GHz, quad core
Motherboard	Gigabyte X58A-UD3R
Harddrive	SanDisk 120GB SSD
Operating System	Windows 8 64-bit

**Figure 39** Hardware specification of the computer running the server

Hardware	Type
RAM	Kingston HyperX Blu DDR3 PC12800/1600MHz CL9 4GB
CPU	Intel Core i3 4130 3,4GHz, dual core
Motherboard	ASRock B85M Pro4
Harddrive	250GB SATA2 HDD
Operating System	Ubuntu 14.04 64-bit (Desktop version)

**Figure 40** Hardware specification of the computer running the client

The web application will be run on the hardware listed in *figure 39*. The Play framework server is started using the **play clean stage** command in the project root and the generated start script executed (Play Framework, n.dc). The test client will be run on the hardware listen in *figure 40*. Both computers are located on the same network, connected trough ethernet trough a NETGEAR WNR3500Lv2 gigabit router.

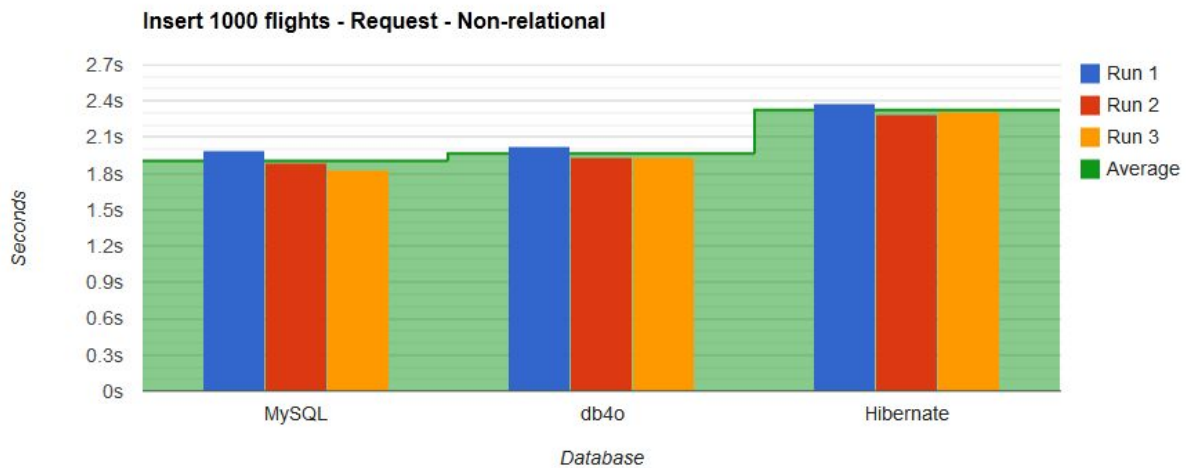
Since the server and client are run from different computers, it is important to ensure the time on both computers match. To ensure this, an NTP server has been configured on the client computer (*figure 40*) and the server computer (*figure 39*) has been made to sync with it before each test run. Since both computers are on the same local gigabit network, NTP should be fairly accurate since the roundtrip between the computers is almost instant (Windl et al., 2006).

Prior to tests being run, all applications and processes not required for running the operating system were terminated to make sure the client and server could get the most out of the system.

The first tests that were run was on the non-relational version, with insertion of flights and selection of flights. At this level, datasets of 1000, 10 000 and 20 000 flights were used. Each test was

repeated three times to be able to calculate an average of the results. By repeating the tests we can see if there was any issues during a test more easily. The plan was to do tests at 80 000 flights as well but it was deemed that the tests would take too long time to complete (the insertion of 80 000 flights took around 3.5 hours to complete in the relational database, and it was estimated that all if both the insert test and select test took the same amount of time on every database the complete test would take 21 hours for a single repetition, 63 hours for three repetitions).

The first tests to be made was on the non-relational version. This version is the closest to the study made by Roopak et al. (2013) where we test only a single table/object. The connection pool on the server was configured to hold ten connections open for the MySQL and Hibernate tests.

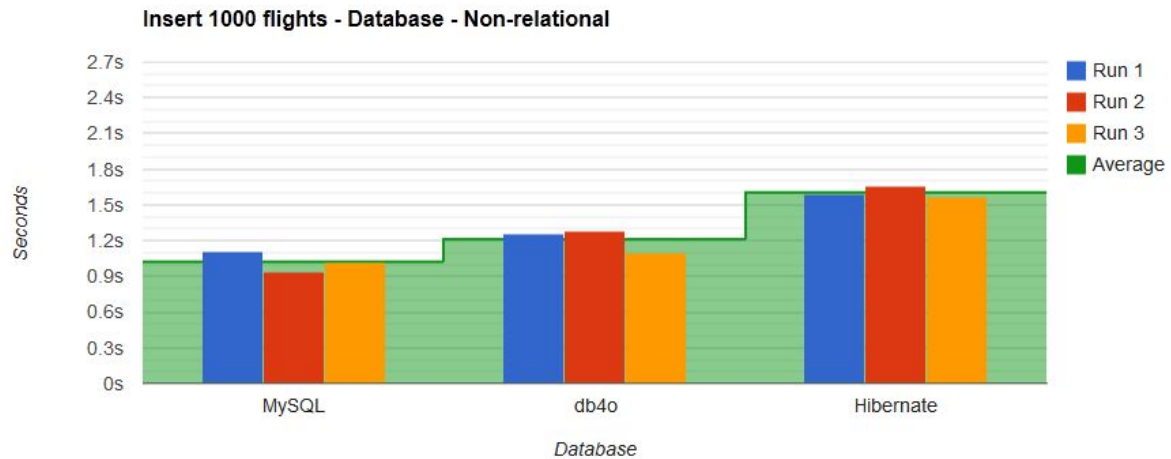


**Figure 41** Response time of inserting 1000 flights using the non-relational version

	Run 1	Run 2	Run 3	Average
MySQL	1.986s	1.893s	1.835s	1.904666667s
db4o	2.02s	1.937s	1.937s	1.964666667s
Hibernate	2.383s	2.283s	2.308s	2.324666667s

**Figure 42** Data behind the chart in *figure 41*

The first test that was done was to insert 1000 flights using the non-relational version of the developed software. As can be seen in *figure 41* and *figure 42*, MySQL and db4o performed almost equal to each other, while Hibernate was a little bit slower, but still quite close to MySQL and db4o. Optimally since this graph includes the whole request time, the time spent doing non-database operations should be equal for each database, but depending on the load of the computers the non-database operations may vary slightly in execution time.



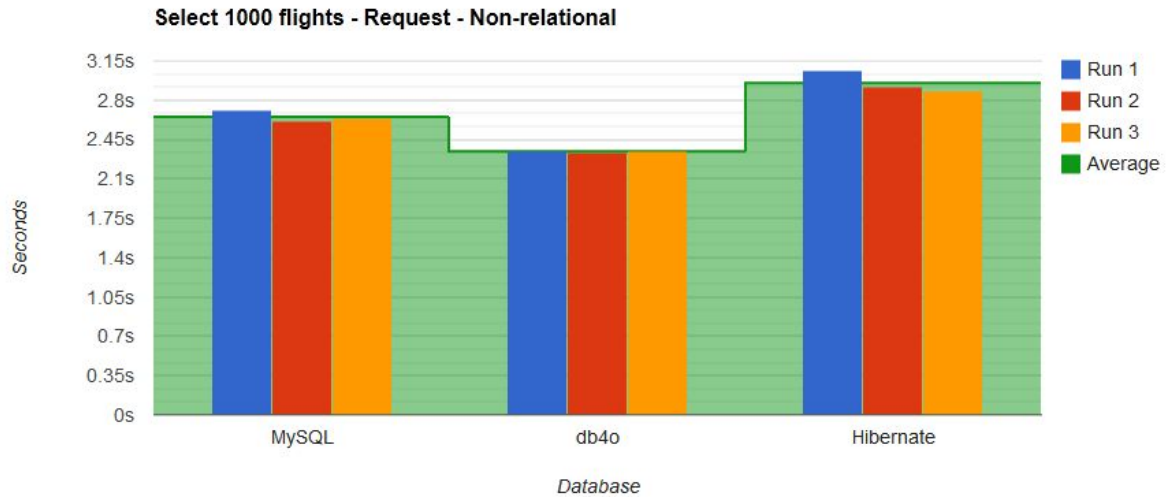
**Figure 43** Time of database operation for the test shown in *figure 41*

	Run 1	Run 2	Run 3	Average	% of request
MySQL	1.11s	0.939s	1.017s	1.022s	53.7%
db4o	1.261s	1.278s	1.096s	1.211666667s	61.7%
Hibernate	1.593s	1.657s	1.565s	1.605s	69.0%

**Figure 44** Data behind the chart in *figure 43*

Figure 43 and figure 44 shows the same test as in the previous figures but is specific to the time spent doing the database operations themselves. Here we can see that MySQL and db4o differ by about 0.190 seconds while MySQL and Hibernate differ by about 0.583 seconds. The smaller difference between MySQL and db4o when looking at the full request time might be attributed to the Play Framework and its web server. The time spent handling the request is roughly between 0.7 - 0.8 seconds in total of 1000 requests, meaning almost one millisecond being spent doing non-database operation, such as transmitting the request, handling the request and transmitting a response. So far our hypothesis is correct that the ORM adds an overhead, where the ORM take around 57% longer than MySQL to perform only the database operation. However, Roopak et al. (2013) claimed that db4o would insert data faster than MySQL, while in our result, db4o is about 19% slower than MySQL.

The database operations in the insert tests takes up about 54 - 69% of the whole request, leaving 31 - 46% to non-database operations.



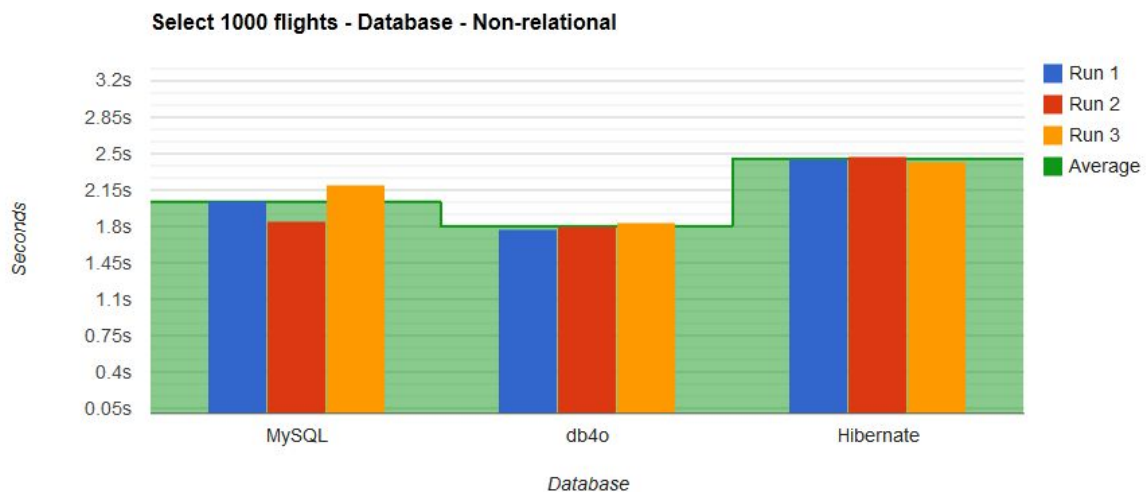
**Figure 45** Response time of selecting 1000 flights using the non-relational version

	Run 1	Run 2	Run 3	Average
MySQL	2.709s	2.611s	2.635s	2.651666667s
db4o	2.349s	2.332s	2.352s	2.344333333s
Hibernate	3.06s	2.92s	2.879s	2.953s

**Figure 46** Data behind the chart in *figure 45*

To no surprise, it took longer to search for data in the database than inserting data. However, we were surprised to see db4o manage to perform slightly better than MySQL, even though Roopak et al. (2013) had found that MySQL performed better than db4o at searching for data in the database. *Figure 47* proves that db4o does indeed perform better than MySQL when looking at the time taken by the database operation.

Yet again Hibernate adds an overhead to MySQL and takes the longest time to perform the test.

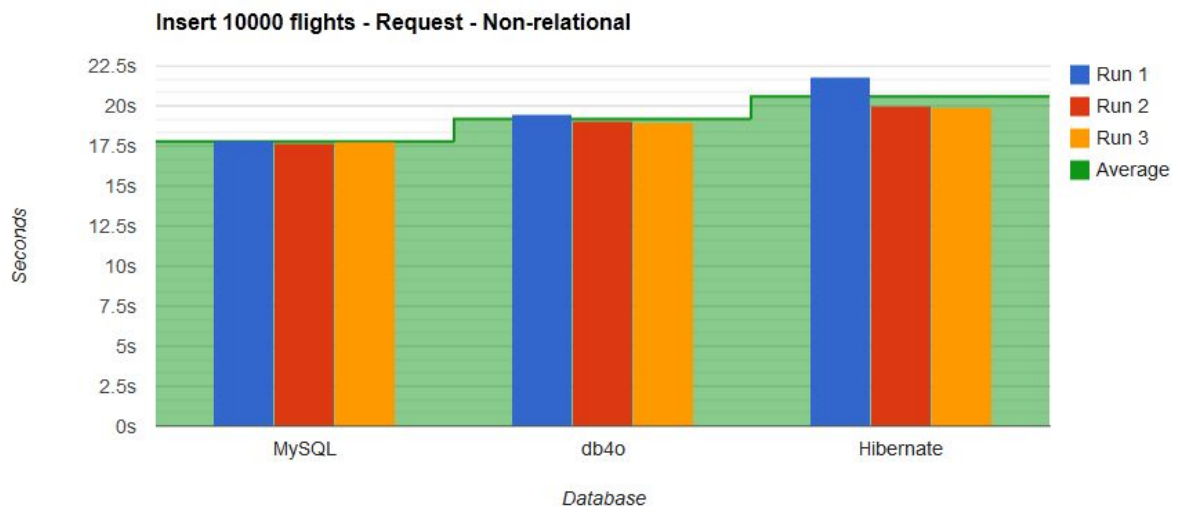


**Figure 47** Time of database operation for the test shown in *figure 45*

	Run 1	Run 2	Run 3	Average	% of request
MySQL	2.043s	1.856s	2.201s	2.033333333s	76.7%
db4o	1.767s	1.793s	1.837s	1.799s	76.7%
Hibernate	2.448s	2.467s	2.424s	2.446333333s	82.8%

**Figure 48** Data behind the chart in *figure 47*

In the select tests each request took around 0.5 - 0.6 seconds of non-database operation. The select tests does not need to de-serialize JSON data from the request as is done in the insert tests. This explains why less time is spent doing non-database operations in the select tests. The database operation took about 77 - 83% of the total request time.



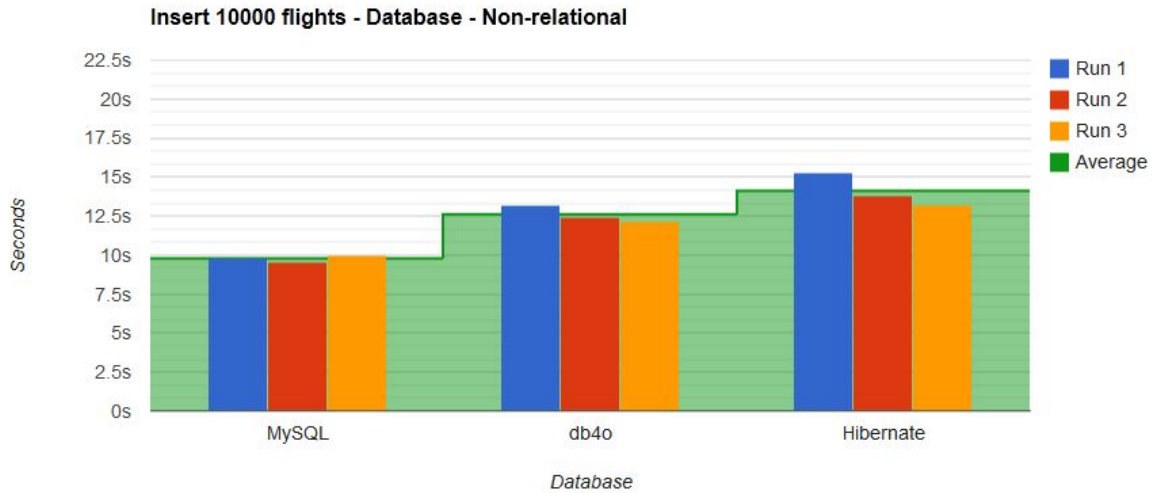
**Figure 49** Response time of inserting 10000 flights using the non-relational version

	Run 1	Run 2	Run 3	Average	Difference 1000 flights
MySQL	17.873s	17.712s	17.748s	17.77766667s	+833.4%
db4o	19.484s	19.069s	18.98s	19.17766667s	+876.1%
Hibernate	21.824s	19.998s	19.927s	20.583s	+785.4%

**Figure 50** Data behind the chart in *figure 49*

Inserting 10 000 flights follows the same trend as when inserting 1000 flights. MySQL performs the best at 17.8 seconds in average, db4o comes second at an average of 19.2 seconds and Hibernate comes last with an average of 20.6 seconds.

When the dataset was increased, so was the total response time. For MySQL and Hibernate the increase was about 9.3 times, and for db4o it took 9.8 times longer. For Hibernate, the increaser was 8.8 times the total response time at 1000 flights. We can see that db4o may not scale as well as MySQL, but the difference is not big enough to be able to draw a certain conclusion that db4o is worse than the other.

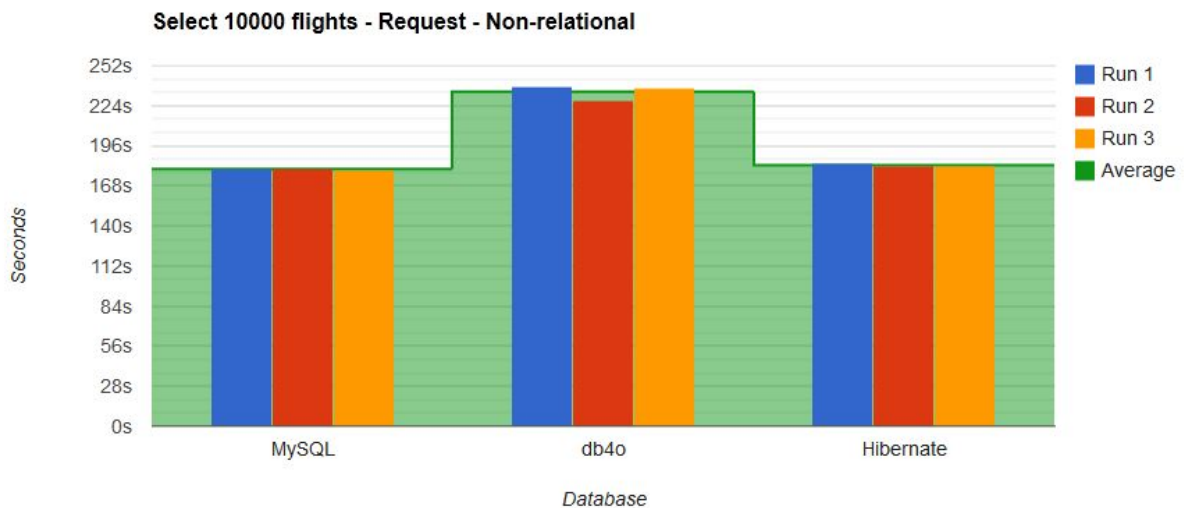


**Figure 51** Time of database operation for the test shown in *figure 49*

	Run 1	Run 2	Run 3	Average	% of request
MySQL	9.798s	9.58s	9.974s	9.784s	55.0%
db4o	13.241s	12.461s	12.153s	12.61833333s	65.8%
Hibernate	15.329s	13.819s	13.189s	14.11233333s	68.6%

**Figure 52** The data behind *figure 51*

The database operations were found to be responsible for around 55 - 69% of the request (*figure 52*), which is almost identical as the test where 1000 flights were inserted. With that one could make an estimate that the non-database operations like transmission time and request handling always take around the same time in the datasets 1000 and 10 000 flights.



**Figure 53** Response time of selecting 10 000 flights using the non-relational version

	Run 1	Run 2	Run 3	Average	% difference to 1000 selects of 1000 flights
MySQL	179.815s	180.03s	179.483s	179.776s	+6680%
db4o	237.586s	227.413s	236.253s	233.7506667s	+9871%
Hibernate	183.458s	182.112s	181.718s	182.4293333s	+6078%

**Figure 54** The data behind the chart in *figure 53*

The select test had quite a large increase in the response time. This is due to several things. Firstly, 10 000 selects are being made, in a dataset of 10 000. This means that not only have the number of selects been increased tenfold, the data to search in have been increased tenfold as well. We will present average time per request at a later stage, where the difference between searching in different datasets will be clearer.

The more data to search in, the longer it takes to find what you are searching for. Since the values being searched does not have a database index, the select time is affected greatly as the dataset increases. The column being searched is not unique, meaning the database has to go through each and every row or object to find all rows or objects that contains that specific value. Had a database index been in use, and/or the column being declared unique, the select times would most likely decrease by far.

We can also see that this time, db4o is the slowest alternative when searching in the database, while it was the fastest at searching in a dataset of 1000 flights. We can conclude that db4o might be faster than MySQL and Hibernate at smaller datasets, but as the dataset grows, MySQL and Hibernate overtakes db4o in performance.



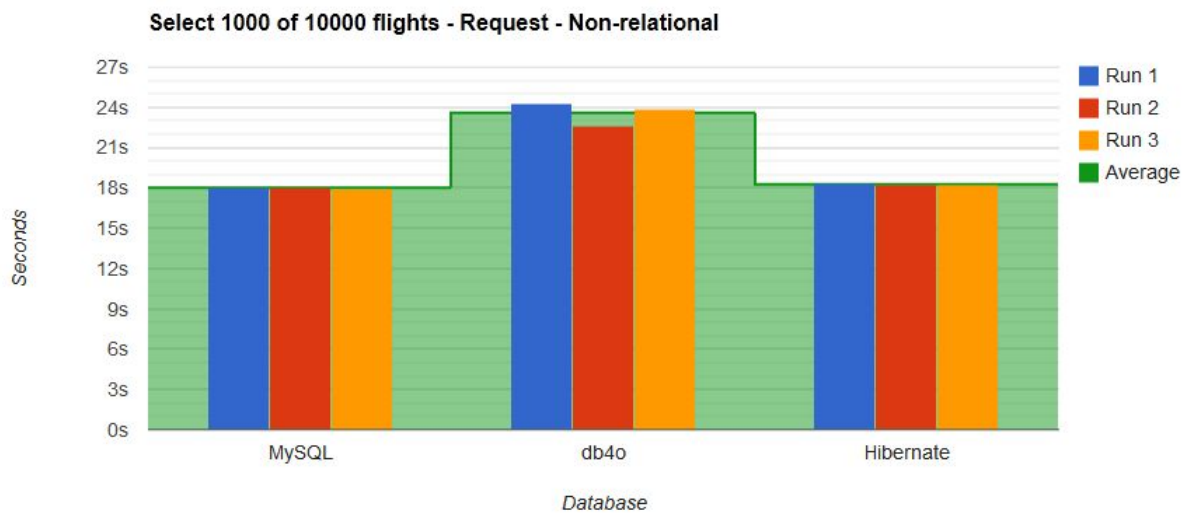
**Figure 55** Time of database operation for test shown in *figure 53*

	Run 1	Run 2	Run 3	Average	% of request
MySQL	173.487s	173.954s	173.41s	173.617s	96.6%
db4o	225.714s	220.152s	224.043s	223.303s	95.5%
Hibernate	179.209s	177.232s	177.135s	177.8586667s	97.5%

**Figure 56** Data behind chart in *figure 55*



Unlike the select test on a dataset of 1000 flights, the database operations stood for almost the whole response time, ranging between 95.5 - 97.5% as can be seen in *figure 56*. This shows how much of a bottleneck an un-optimised database can be to an application, where no consideration to indexes and other performance tweaks have been made.

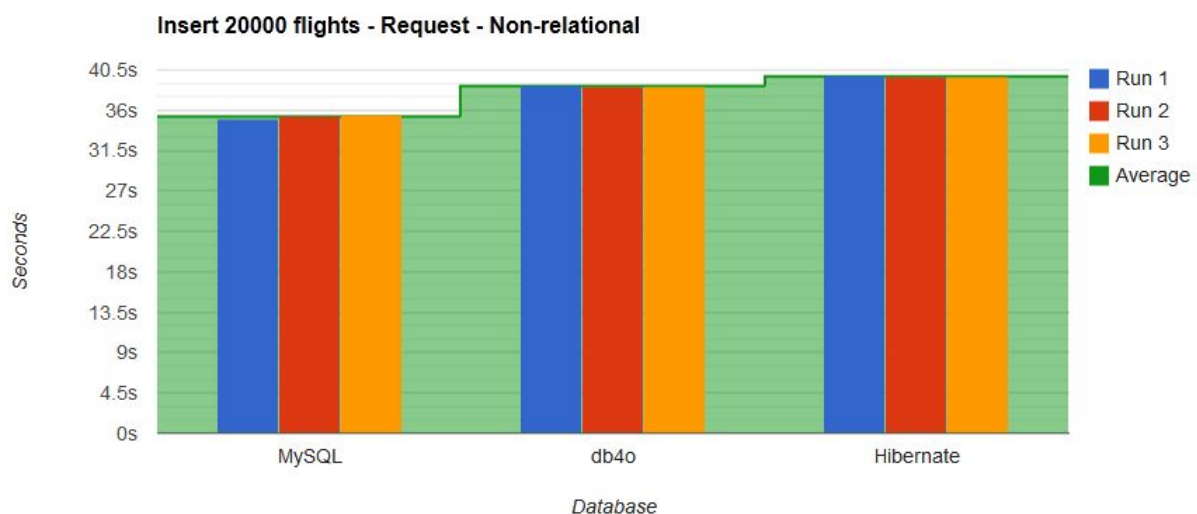


**Figure 57** Select 1000 out of 10 000 flights, total response time using the non-relational version

	Run 1	Run 2	Run 3	Average	Difference 1000 flights
MySQL	18.023s	18.025s	17.963s	18.00366667s	+579.0%
db4o	24.285s	22.61s	23.875s	23.59s	+906.3%
Hibernate	18.392s	18.21s	18.198s	18.26666667s	+518.6%

**Figure 58** Data behind chart in *figure 57*

Graphing the total response time of the first 1000 selects in the dataset of 10 000 flights we can get a result that is easier to compare to the chart in *figure 45*. We can see that the total response time for MySQL was roughly 6.8 times larger when searching 10 000 rows instead of 1000 rows, while Hibernate took roughly 6.2 times longer in the larger dataset. Db4o did not scale as well with a 9 times increase.

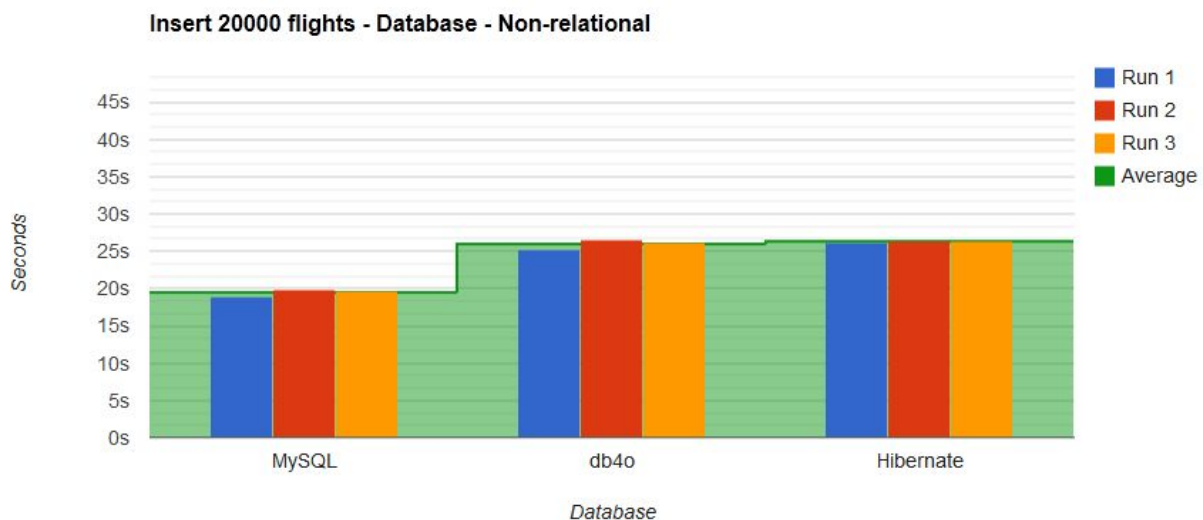


**Figure 59** Insertion of 20 000 flights, total response time using the non-relational version

	Run 1	Run 2	Run 3	Average	Difference 10 000 flights
MySQL	35.032s	35.346s	35.441s	35.273s	+98.4%
db4o	38.839s	38.596s	38.616s	38.68366667s	+98.6%
Hibernate	39.88s	39.706s	39.709s	39.765s	+93.2%

**Figure 60** Data behind the chart in *figure 59*

Doubling the dataset once more to 20 000 flights yields almost a linear increase in total response time compared to 10 000 flights. 20 000 flights took almost exactly twice as long as inserting 10 000 flights into MySQL and db4o. However, Hibernate seems to scale slightly better but the small difference could possibly be attributed to minor changes in the operating systems load. If however Hibernate continues to scale better than MySQL, it may or may not reach very similar response times, but further research needs to be done to establish that. One thing for sure, in theory, is that Hibernate should never be able to outperform MySQL, unless the SQL Hibernate generates is more optimised and efficient than the manually written SQL.



**Figure 61** Total time of database operations, inserting 20 000 flights using the non-relational version

	Run 1	Run 2	Run 3	Average	% of request
MySQL	18.925s	19.861s	19.636s	19.474s	55.2%
db4o	25.226s	26.531s	26.122s	25.95966667s	67.1%
Hibernate	26.244s	26.358s	26.377s	26.32633333s	66.2%

**Figure 62** Data behind chart in *figure 61*

Again the database operation takes up roughly the same part of the request (*figure 62*) as in the test with 1000 and 10 000 flights. It is pretty safe to conclude that the Play framework and its built-in web server performs quite consistently.

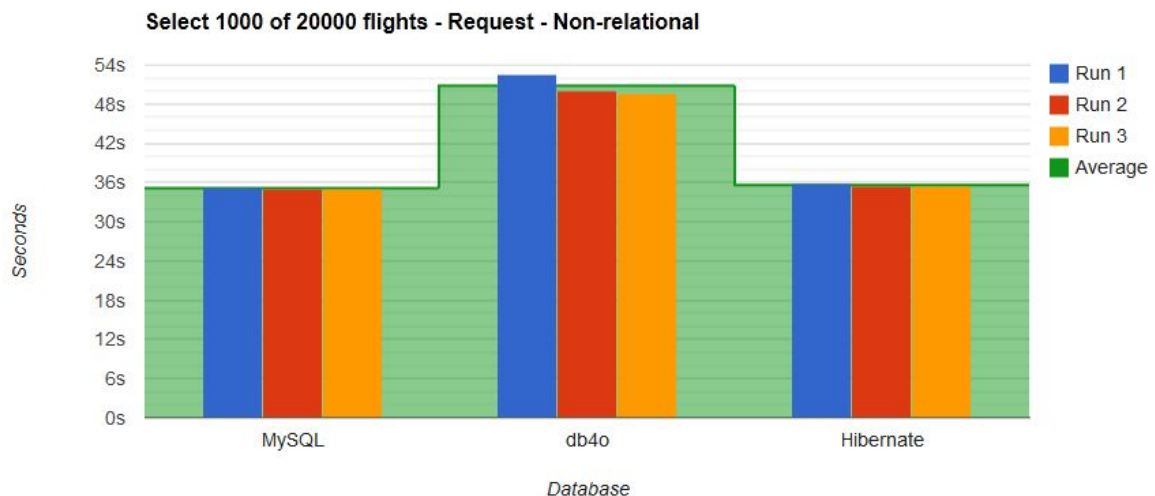


**Figure 63** Total response time to search 20 000 times in a dataset of 20 000 flights using the non-relational version

	Run 1	Run 2	Run 3	Average	% difference to 10 000 selects of 10 000 flights
MySQL	702.969s	698.621s	701.37s	700.9866667s	+289.9%
db4o	1036.685s	1012.645s	997.073s	1015.467667s	+334.4%
Hibernate	708.588s	709.933s	707.46s	708.6603333s	+288.5%

**Figure 64** Data behind the chart in *figure 63*

At 20 000 searches in a dataset of 20 000 flights, db4o continues to perform worse than the other alternatives and the difference between db4o and MySQL continues to grow larger. It is getting more obvious that without proper optimisation, doing some heavy searching will not yield fast results. Looking at the graph like this one could believe it is extremely slow in a web environment, however, it might only be a dozen milliseconds per request, which will be presented later in this report. A dozen milliseconds could mean a lot though depending on the traffic you are getting and how long each user is expected to wait.

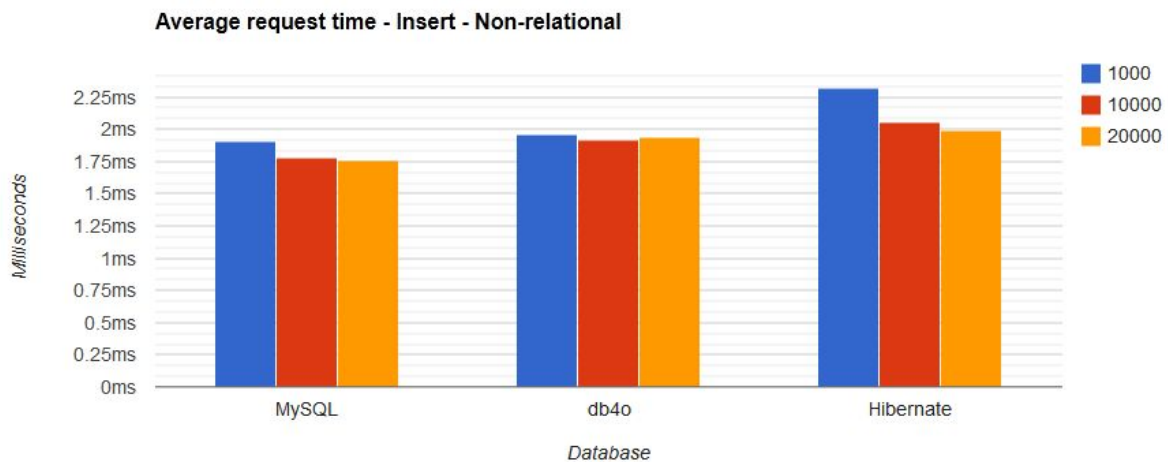


**Figure 65** Select 1000 out of 20 000 flights, total response time using the non-relational version

	Run 1	Run 2	Run 3	Average	Difference 20000 flights
MySQL	35.329s	34.961s	35.106s	35.132s	+95.1%
db4o	52.626s	50.069s	49.655s	50.78333333s	+115.3%
Hibernate	35.908s	35.534s	35.389s	35.61033333s	+94.9%

**Figure 66** Data behind chart in *figure 65*

Taking the first 1000 collected results we can see that MySQL and Hibernate took almost twice as long compared to the previous dataset of 10 000 flights, when the dataset has been doubled to 20 000 flights. Db4o however increased a bit over twice the previous dataset.



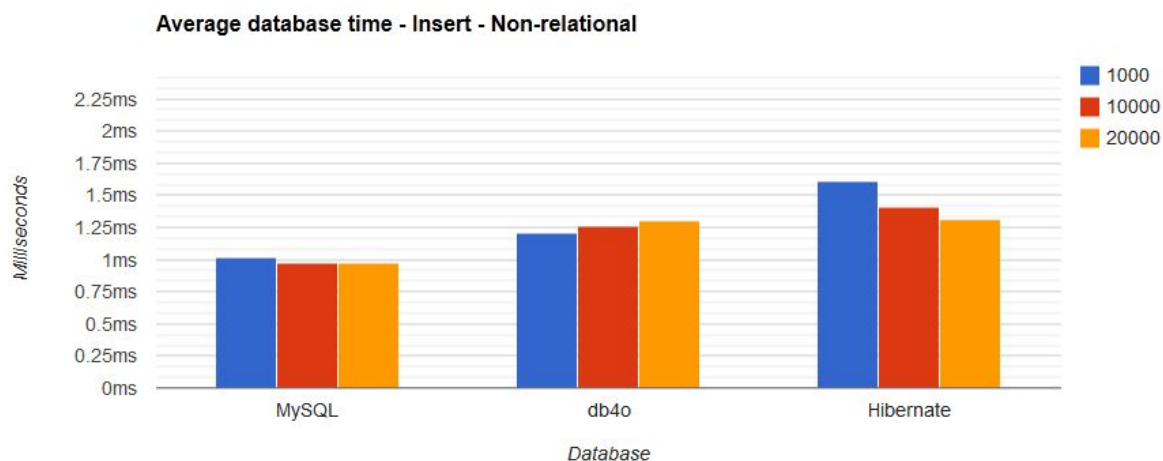
**Figure 67** Average response time for 1000, 10 000 and 20 000 insert requests using the non-relational version

	1000	10000	20000
MySQL	1.904666667ms	1.777766667ms	1.76365ms
db4o	1.964666667ms	1.917766667ms	1.934183334ms
Hibernate	2.324666667ms	2.0583ms	1.98825ms

**Figure 68** Data behind chart in *figure 67*

The average request time when inserting data is quite interesting. When inserting data, the size of the database does not seem to affect the request time very much when an average is computed across all the requests.

Note how the average for 1000 requests are higher than for 10 000 and 20 000 requests. This could be due to high values affecting the average to a greater degree since the test size is smaller. It could also be due to variances when the tests were run, as the tests with 1000, 10 000 and 20 000 was run at separate times.



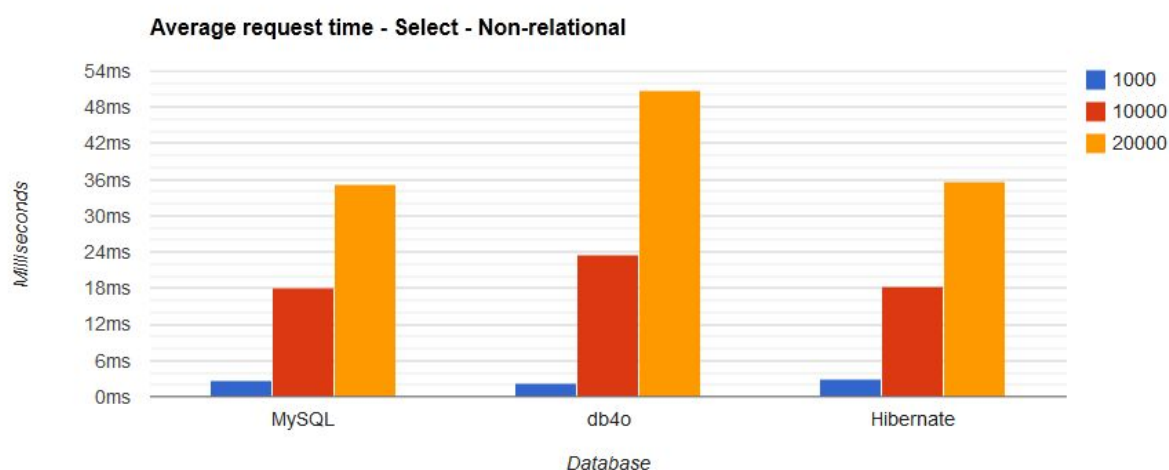
**Figure 69** Average database time for 1000, 10 000 and 20 000 insert requests using the non-relational version

	1000	10000	20000
MySQL	1.022ms	0.9784ms	0.9737ms
db4o	1.211666667ms	1.261833333ms	1.297983334ms
Hibernate	1.605ms	1.411233333ms	1.316316667ms

**Figure 70** Data behind chart in figure 69

MySQL is close to average 1 millisecond per insert in all tests. Db4o comes close at about 0.2 milliseconds slower than MySQL. Hibernate does not perform as well, coming in around 0.3 - 0.6 milliseconds slower than MySQL.

We can observe here too that the average on 1000 rows is a little bit higher than 10 000 and 20 000 rows. Db4o however increases slightly as the dataset grows. This could show that inserts are getting slower as there are more data in the database, though the increase is very small and further tests would need to be done—with larger datasets—to make sure that is the case.

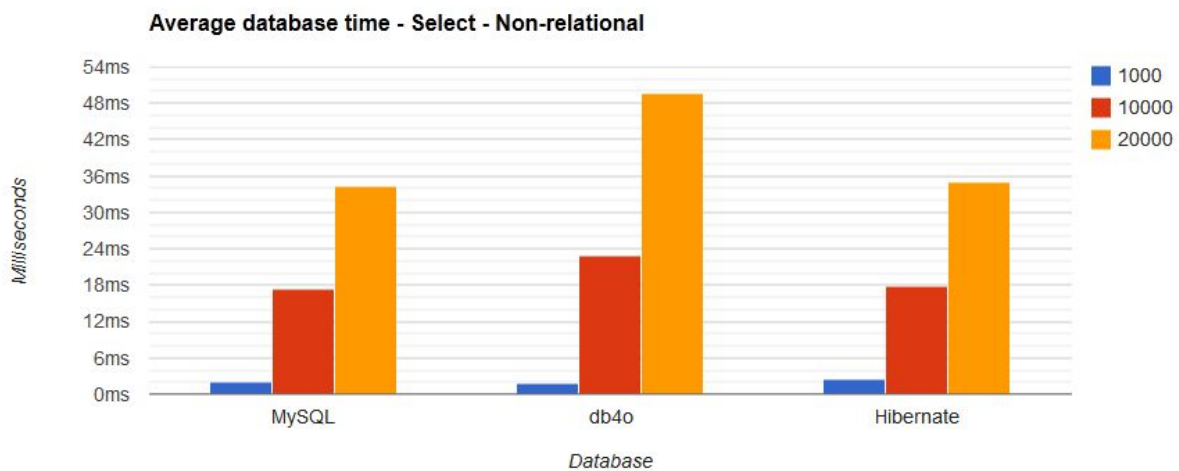


**Figure 71** Average response time for 1000 searches in a dataset of 1000, 10 000 and 20 000 flights using the non-relational version

	1000	10000	20000
MySQL	2.651666667ms	18.00366667ms	35.132ms
db4o	2.344333333ms	23.59ms	50.78333333ms
Hibernate	2.953ms	18.26666667ms	35.61033333ms

**Figure 72** Data behind the chart in *figure 71*

When calculating the average request time for searching the database, only the 1000 first requests have been averaged for each dataset. The reason is that only 1000 searches were made on the dataset with 1000 flights, and therefore it would not be possible to compare the average of 10 000 searches in a dataset of 10 000 flights, since it will most likely result in a lower average (refer to *figure 67* and the discussion below it regarding lower average the more requests have been averaged).



**Figure 73** Average response time for 1000 searches in a dataset of 1000, 10 000 and 20 000 flights using the non-relational version

	1000	10000	20000
MySQL	2.033333333ms	17.347ms	34.381ms
db4o	1.799ms	22.77466667ms	49.55533333ms
Hibernate	2.446333333ms	17.83633333ms	35.011ms

**Figure 74** Data behind the chart in *figure 73*

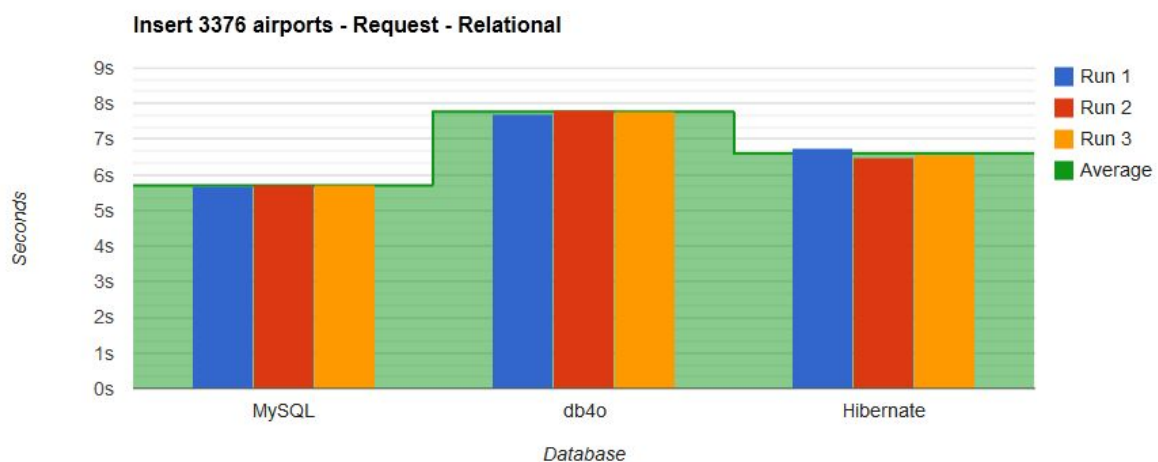
We can see that as the dataset to search increases, the average database time increases as well. When searching in the MySQL database, an average search took about 2 milliseconds in a dataset of 1000 flights. In a 10 times large dataset an average request took 17.3 milliseconds. That is an increase by 753%, or 8.53 times larger than the average request to search in a dataset of 1000 flights. The average request time to search in a dataset of 20 000 flights is 34.381 milliseconds, an 98% increase from 10 000 flights. The request time seems to be closer to linear between a dataset of 20 000 and 10 000 where the increase was nearly double, but the increase from 1000 flights to 10 000 flights was only 8.53 times larger, instead of 10 times larger.

As previously stated, db4o seems to perform slightly better than MySQL and Hibernate in the dataset of 1000 flights. However, it quickly becomes a lot slower as the dataset grows. The average request time between 1000 and 10 000 flights increases 12.7 times, while the average request time between 10 000 and 20 000 flights increases by 2.18 times. This means that in our tests, when the dataset has been doubled, the average request time is almost doubled as well, making it almost

linear increase depending on the size of the dataset. This makes sense if the database visit each object in roughly the same time when comparing the search term to the object. The difference between 1000 and 10 000 flights are over 10 times as large though, which leads us to believe that there is some added overhead when the dataset is increased.

Hibernate, to our surprise does not seem to add very much overhead to MySQL as we thought, but it could still be considerable when doing a lot of database operations. The average of requests in a dataset of 20 000 flights differs 0.63 milliseconds between MySQL and Hibernate, this is the overhead. However, when doing a lot of operations, this quickly adds up. If you were to do 20 000 of this operation, Hibernate would be 12.6 seconds slower than MySQL, following the average.

Next up is the tests on the relational version, where an airport table and object have been created and flights linked to these airports. For these tests, the connection pool on the server was configured with 150 connections after problems running the relational tests, where occasionally the pool was emptied and requests would get stuck for several minutes at times.



**Figure 75** Insertion of 3376 airports, total response time using the relational version

	Run 1	Run 2	Run 3	Average
MySQL	5.674s	5.702s	5.727s	5.701s
db4o	7.695s	7.82s	7.792s	7.769s
Hibernate	6.76s	6.499s	6.545s	6.601333333s

**Figure 76** Data behind chart in *figure 75*

The airport test was run each repetition before the other tests, and was only tested at a dataset of 3376 airports. Since the airports are part of a relation with flights, all airports had to be present in the database to be able to insert the flights, in case one of the flights would reference an airport that had not been inserted into the database, an error would have occurred.

We can see in *figure 75* and *figure 76* that MySQL is the fastest to insert all airports when looking at the response time. Hibernate is roughly 0.9 seconds slower than MySQL, and db4o is roughly 2 seconds slower than MySQL on average.



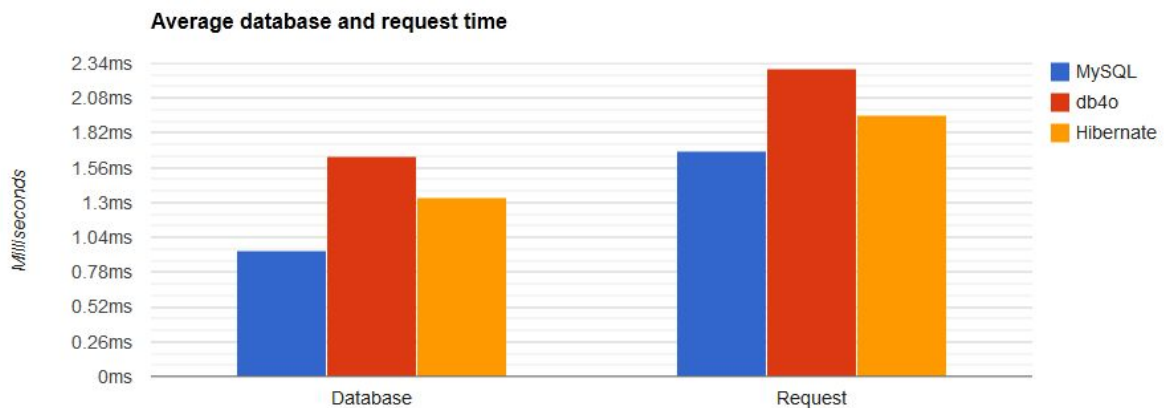


**Figure 77** Insertion of 3376 airports, total time of database operations

	Run 1	Run 2	Run 3	Average	% of request
MySQL	3.201s	3.145s	3.218s	3.188s	55.9%
db4o	5.438s	5.752s	5.528s	5.572666667s	71.7%
Hibernate	4.883s	4.289s	4.397s	4.523s	68.5%

**Figure 78** Data behind the chart in *figure 77*

The insert test continues to be consistent with the previous insert tests, taking about 56 - 72% of the request.



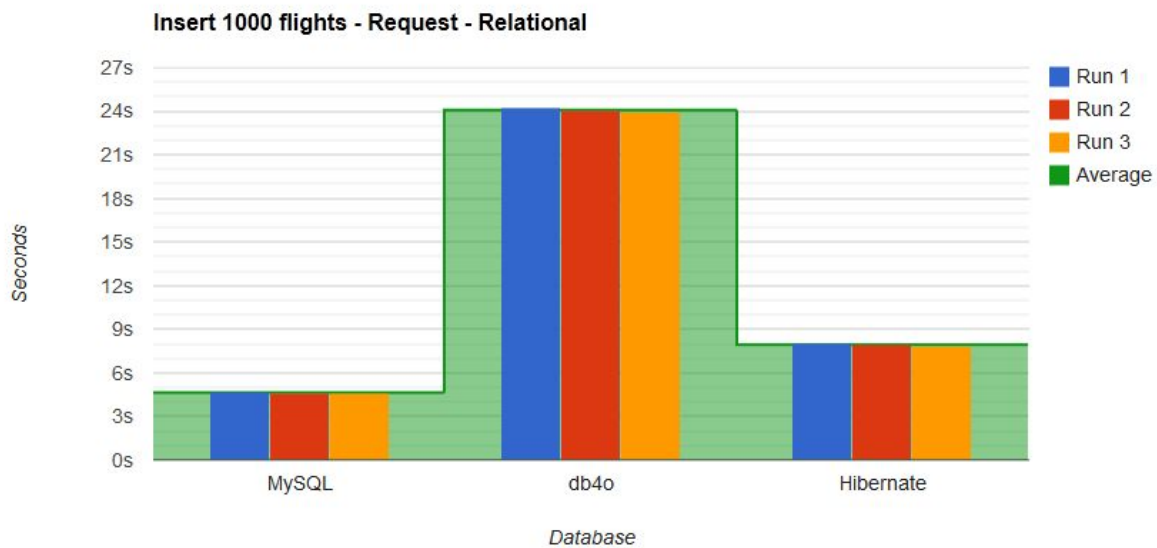
**Figure 79** Average database and response time when inserting 3376 airports using the relational version

	MySQL	db4o	Hibernate
Database	0.9443127962ms	1.650671406ms	1.339751185ms
Request	1.688684834ms	2.301244076ms	1.955371248ms

**Figure 80** Data behind chart in *figure 79*

When calculating the average request time and the average database time of the 3376 airports, we can clearly see that MySQL is the winner at 0.9 milliseconds per database insert on average. Hibernate is about 0.4 milliseconds slower than MySQL per database insert, and db4o is about 0.7 milliseconds slower than MySQL per database insert.





**Figure 81** Total response time to insert 1000 flights

	Run 1	Run 2	Run 3	Average
MySQL	4.703s	4.637s	4.631s	4.657s
db4o	24.225s	24.073s	23.956s	24.08466667s
Hibernate	8.012s	7.966s	7.849s	7.942333333s

**Figure 82** Data behind chart in figure 81

Db4o loses a lot when it comes to tests where each flight links to two airports (origin and destination). When a flight is received from the client, they contain two airport objects where only the IATA code is provided. A search in the database has to be made to retrieve the real airport object and replace the object from the request. Two lists containing departing and arriving flights have to be updated with the received flight and stored again. This whole process seems to be very time expensive for db4o.

```

1 | ObjectSet result = db.queryByExample(flight.getDest());
2 | Airport destAirport = (Airport) result.next();
3 |
4 | destAirport.getFlyingIn().add(flight);
5 | flight.setDest(destAirport);
6 |
7 | result = db.queryByExample(flight.getOrigin());
8 | Airport originAirport = (Airport) result.next();
9 |
10 | originAirport.getFlyingOut().add(flight);
11 | flight.setOrigin(originAirport);
12 |
13 | db.store(flight);
14 | db.store(destAirport);
15 | db.store(originAirport);
16 | db.commit();

```

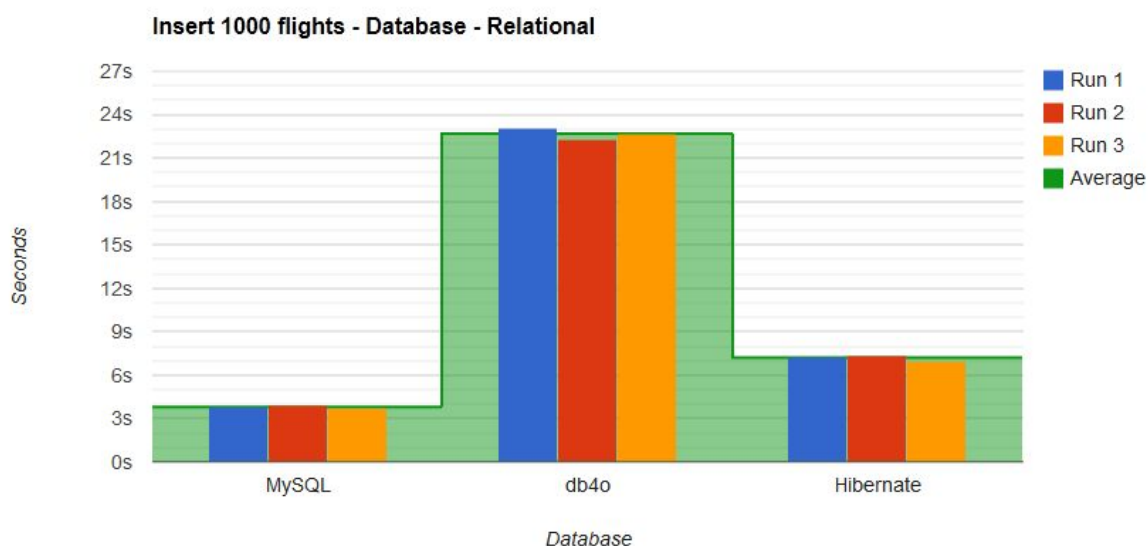
**Figure 83** Insertion of a flight into the object database

Figure 83 shows the process of inserting the flight into the database. After the flight has been stored, the two airports need to be stored again since their lists of arriving and departing flights have been updated.

There might be a better and faster way to do this process and probably a lot of optimisations and

tweaks that can be done but that would require a more deeper understanding and knowledge about db4o.

Hibernate also adds a considerable overhead to MySQL, being 3.285 seconds slower than MySQL on average.

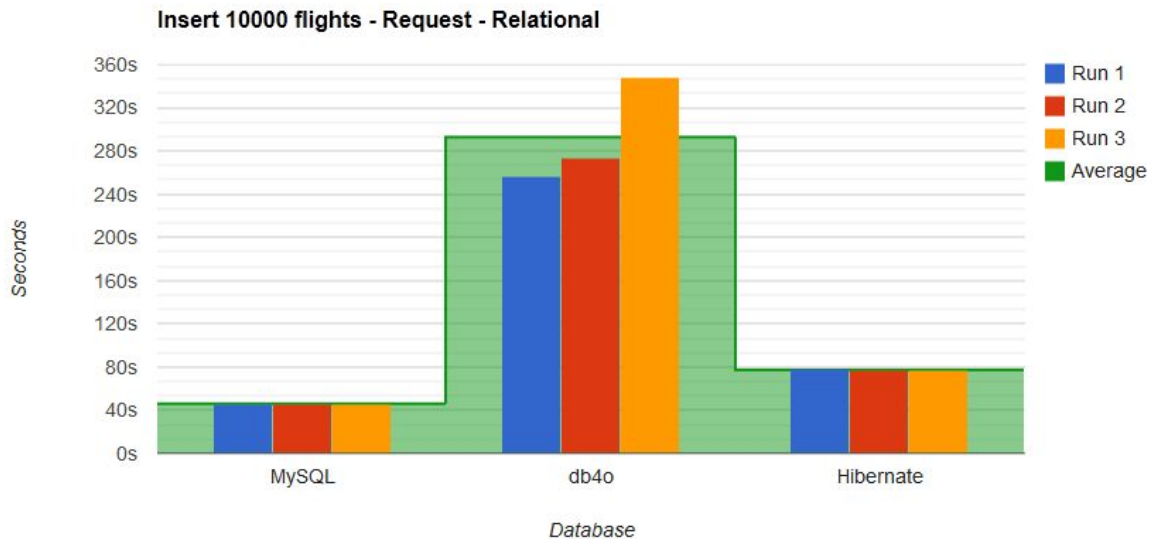


**Figure 84** Total database time to insert 1000 flights

	Run 1	Run 2	Run 3	Average	% of request
MySQL	3.853s	3.881s	3.686s	3.806666667s	81.7%
db4o	23.054s	22.249s	22.722s	22.675s	94.1%
Hibernate	7.294s	7.359s	6.987s	7.213333333s	90.8%

**Figure 85** Data behind chart in *figure 84*

Compared to the non-relational insert tests, the relational insert test seems to take up larger part of the total response time. This is mostly due to more database operations being performed, as previously stated. As more database operations are performed, more time is spent on the database operation, while the same amount of time should be spent handling non-database operations as in the non-relational version.



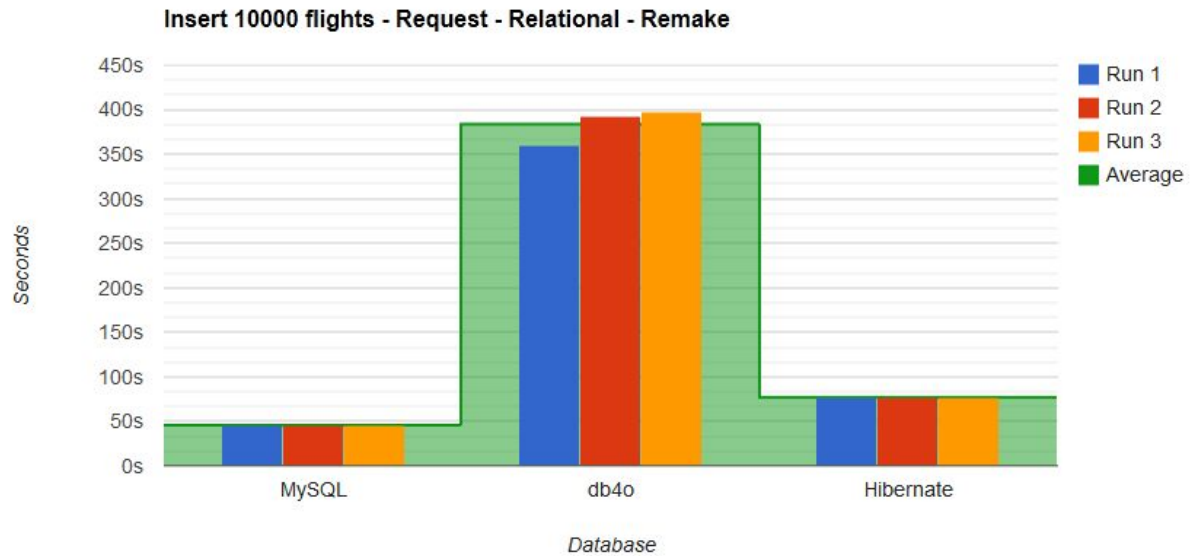
**Figure 86** Total response time to insert 10 000 flights

	Run 1	Run 2	Run 3	Average	% difference to 1000 flights
MySQL	46.351s	45.983s	45.891s	46.075s	+889.4%
db4o	256.285s	273.58s	348.181s	292.682s	+1115.2%
Hibernate	77.904s	77.323s	77.054s	77.427s	+874.9%

**Figure 87** Data behind chart in figure 86

At 10 000 inserted flights, db4o starts to perform very inconsistently. At first, we believed something was wrong with the computer or router that would affect the test so the test was remade, but with the same results. It is unclear why the response time varies so much.

One theory we have might be in regard to the maximum heap space the Java Virtual Machine is allocated. We noticed that db4o used considerably more RAM than MySQL and Hibernate, where the entire server application would use around 300MB of RAM during insert tests using MySQL and Hibernate, it would use around 1300MB RAM when using db4o. Our theory is that the garbage collector ran during the db4o test as the Java max heap space was approached, which slowed the test down. We tried to increase the heap space to 8096MB (which gave us 7759MB RAM that Java will attempt to use, according to `runtime.maxMemory()`) and remake the test once more.

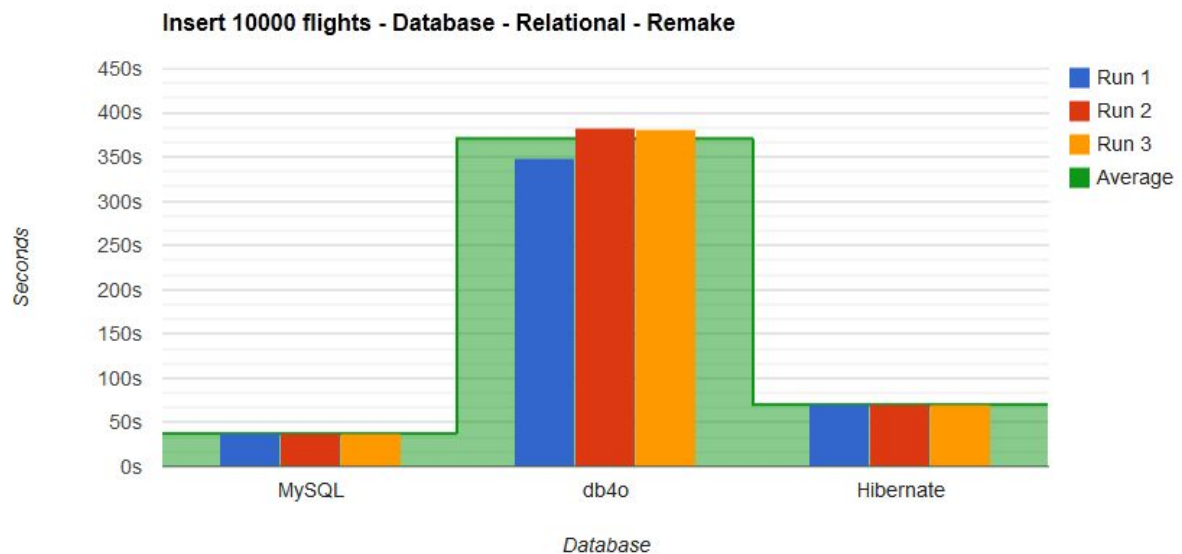


**Figure 88** Remake of the test in *figure 86*

	Run 1	Run 2	Run 3	Average	% difference to 1000 flights
MySQL	46.176s	45.729s	45.781s	45.89533333s	+885.5%
db4o	359.503s	393.513s	398.33s	383.782s	+1496.5%
Hibernate	77.29s	77.002s	77.22s	77.17066667s	+871.6%

**Figure 89** Data behind the chart in *figure 88*

The remake is not any better. There is still a quite large difference between the first and second run of db4o, not to mention that each run took even longer than the average in the original test. This means that different sizes of the heap space did not seem to affect the inconsistent response times, but may have affected the overall performance of db4o since an increased average of almost 100 seconds.



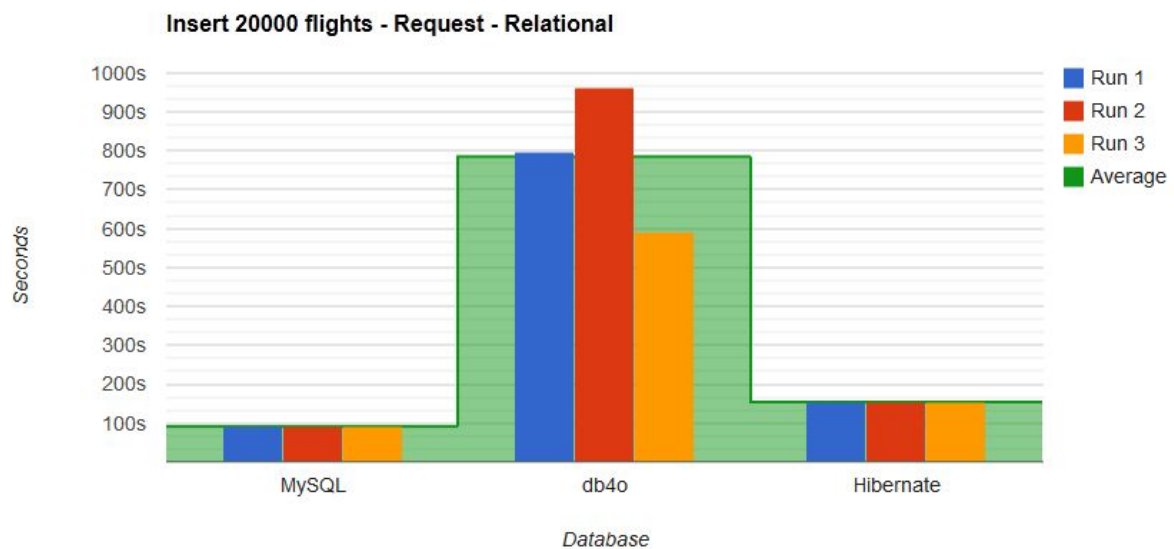
**Figure 90** Total database time to insert 10 000 flights

	Run 1	Run 2	Run 3	Average	% of request
MySQL	37.545s	37.119s	37.576s	37.41333333s	81.5%
db4o	348.692s	382.227s	381.556s	370.825s	96.6%
Hibernate	70.448s	70.363s	69.951s	70.254s	91.0%

**Figure 91** Data behind chart in *figure 90*

Another theory was that it was the server application that was misbehaving, but looking at the results in *figure 90* and *figure 91* we can see that there are large differences between the first and second run of db4o even on the database level.

Hibernate continues to add an overhead to MySQL when inserting data. While Hibernate was 89% slower than MySQL to insert 1000 flights, it was 88% slower than MySQL to insert 10 000 flights. So far, it seems like the overhead Hibernate adds to MySQL is near 90%.



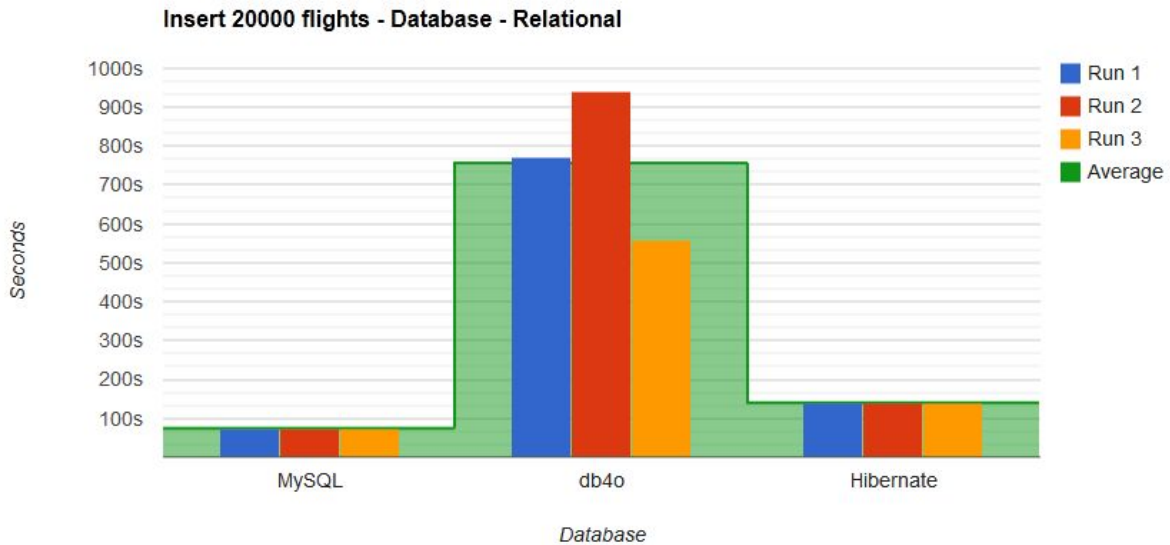
**Figure 92** Total response time to insert 20 000 flights

	Run 1	Run 2	Run 3	Average	% difference to 10 000 flights
MySQL	91.571s	91.617s	91.526s	91.57133333s	+99.5%
db4o	798.345s	962.295s	592.875s	784.505s	+104.4%
Hibernate	154.346s	153.914s	154.715s	154.325s	+100.0%

**Figure 93** Data behind the chart in *figure 92*

The dataset has been doubled and so has the average total response time as well.

Db4o continues to have inconsistent results. At 20 000 flights, the difference between each run is even larger than at 10 000 flights. Since this result have now been observed in two different tests and in several test runs and seems to only happen to db4o, it is possible that the performance of db4o is inconsistent and not very predictable when it comes to more complex objects than those found in the non-relational version.



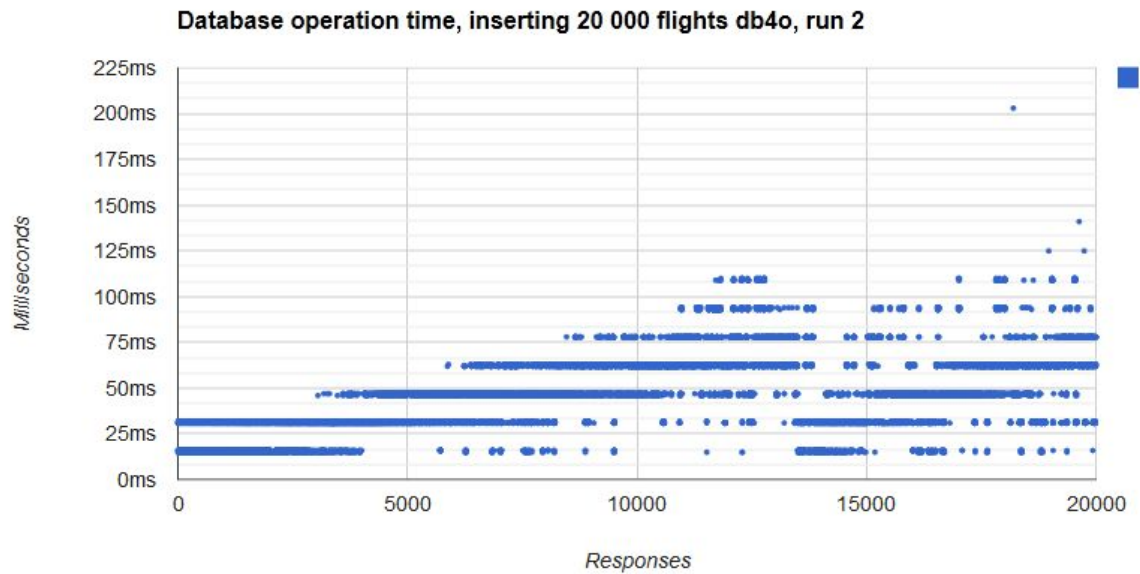
**Figure 94** Total database time to insert 20 000 flights

	Run 1	Run 2	Run 3	Average	% of request
MySQL	74.994s	74.871s	75.116s	74.99366667s	81.9%
db4o	771.782s	939.34s	556.611s	755.911s	96.4%
Hibernate	140.957s	139.854s	141.235s	140.682s	91.2%

**Figure 95** Data behind the chart in *figure 94*

We can see that once more, the differences between the different runs of db4o shows up.

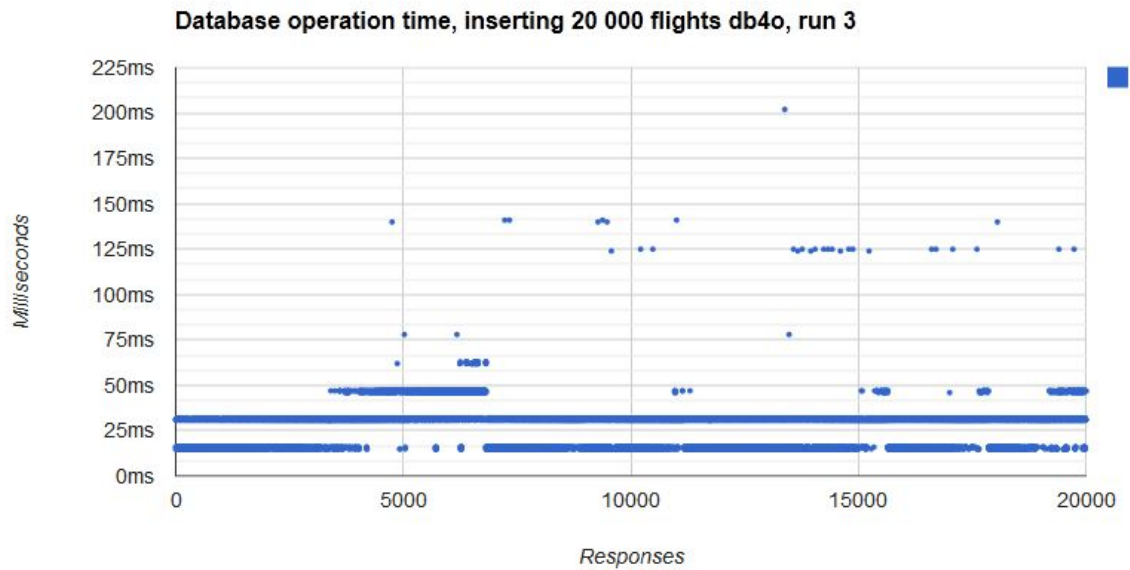
MySQL continues to be the fastest database to insert the flights, while Hibernate has a 88% overhead on average. From the tests so far we can conclude that Hibernate seems to have an overhead of about 90% compared to MySQL at this specific task.



**Figure 96** A scatter chart of the 20 000 responses gathered during the second run of the db4o test, plotting the database operation time for each response

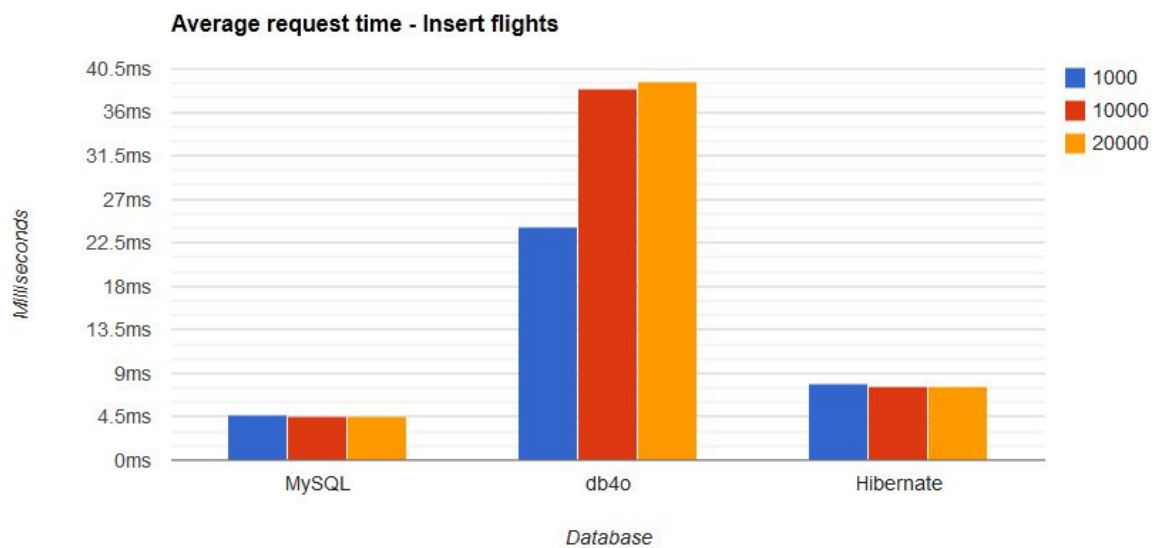
A scatter chart was made in an attempt to understand the problem behind db4o a bit better. Note that the order of the points are the order they are stored in the database and not the order they are received from the server. This is due to responses being saved to a result database by several threads concurrently.

We can see that at the start of the test, database operations alternates between around 15 and 30 milliseconds. After a while it starts to alternate between 30 and 50 milliseconds instead, occasionally hitting 15 milliseconds. As the test progresses, some database operations start to take longer and longer. Most of the database operations seems to happen between 15 and 115 milliseconds, with a couple of requests taking longer than that. However, it does not explain why we get different results each run.



**Figure 97** A scatter chart of the 20 000 responses gathered during the third run of the db4o test, plotting the database operation time for each response

A scatter chart was made for the third run as well, the run with the lowest execution time. Here we can see the difference, that a majority of the database operations are between 15 and 30 milliseconds, occasionally hitting 50 milliseconds and rarely hitting above.



**Figure 98** Average response time when inserting flights

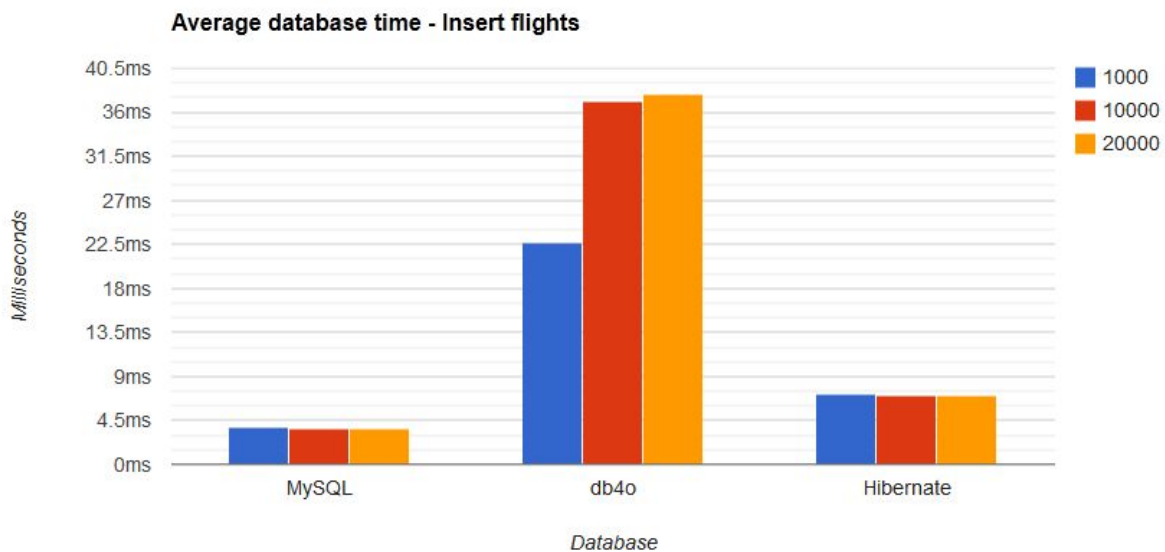
	1000	10000	20000
MySQL	4.657ms	4.589533333ms	4.578566667ms
db4o	24.08466667ms	38.3782ms	39.22525ms
Hibernate	7.942333333ms	7.717066667ms	7.71625ms

**Figure 99** Data behind chart in *figure 98*



We can see some similarities to the non-relational version when looking at the average response time for MySQL and Hibernate, namely that they are about the same for each dataset. There are not much of a change other than what have been previously stated that the average goes down slightly as the dataset increases.

Db4o however had quite some difference this time around. Db4o seems to be a lot slower than the rest at inserting data, and as the dataset increases, so does the average as well. The increase between 1000 and 10 000 is pretty steep, with an increase of 14 milliseconds. In the previous tests we could see that db4o had a pretty consistent result at the 1000 flights dataset, but at 10 000 and 20 000 the result became very inconsistent and is likely the cause for the increasing average. The early conclusion is that db4o cannot handle large amounts of data as good as MySQL.



**Figure 100** Average database operation time, inserting flights

	1000	10000	20000
MySQL	3.806666667	3.741333333	3.749683333
db4o	22.675	37.0825	37.79555
Hibernate	7.213333333	7.0254	7.0341

**Figure 101** Data behind the chart in *figure 100*

From the table in *figure 101* we can see that the non-database operations takes roughly 0.8 milliseconds out of the full request when inserting into MySQL, while roughly 0.7 milliseconds when inserting using Hibernate. In the db4o tests, non-database operations seems to take around 1 - 2 milliseconds per request.

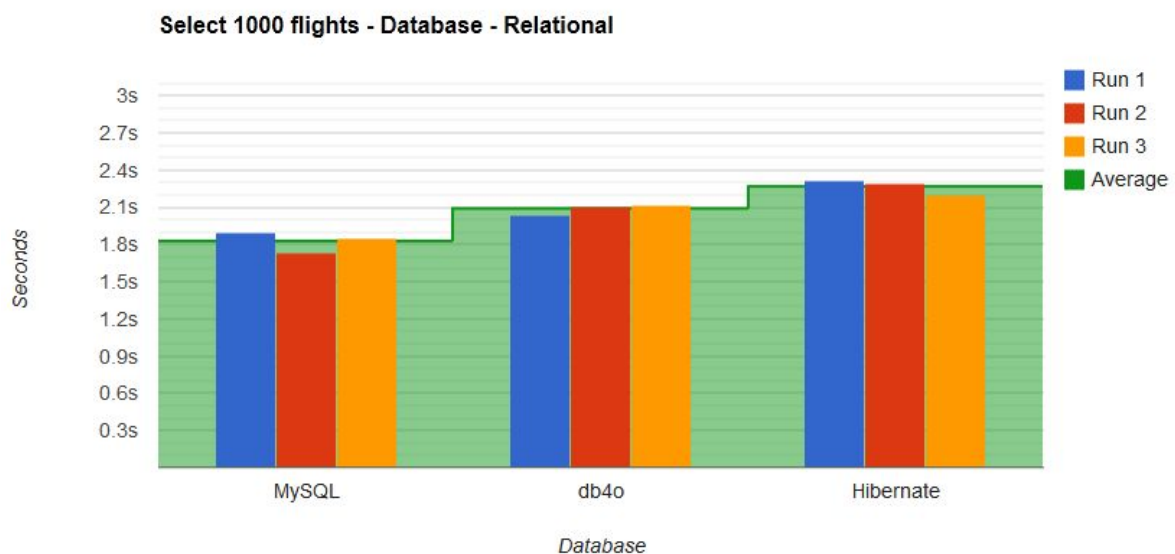


**Figure 102** Total response time to do 1000 selects of 1000 flights

	Run 1	Run 2	Run 3	Average
MySQL	2.421s	2.399s	2.409s	2.409666667s
db4o	2.475s	2.538s	2.453s	2.488666667s
Hibernate	2.745s	2.712s	2.661s	2.706s

**Figure 103** Data behind the chart in *figure 102*

There is not a big difference between the different databases and the ORM framework when doing 1000 selects in a database with 1000 flights. The difference between the average of MySQL and the average of db4o is 80 milliseconds. At this point it does not really matter which database you use unless those 80 milliseconds are important. Hibernate is a bit slower than that, by about 220 milliseconds.

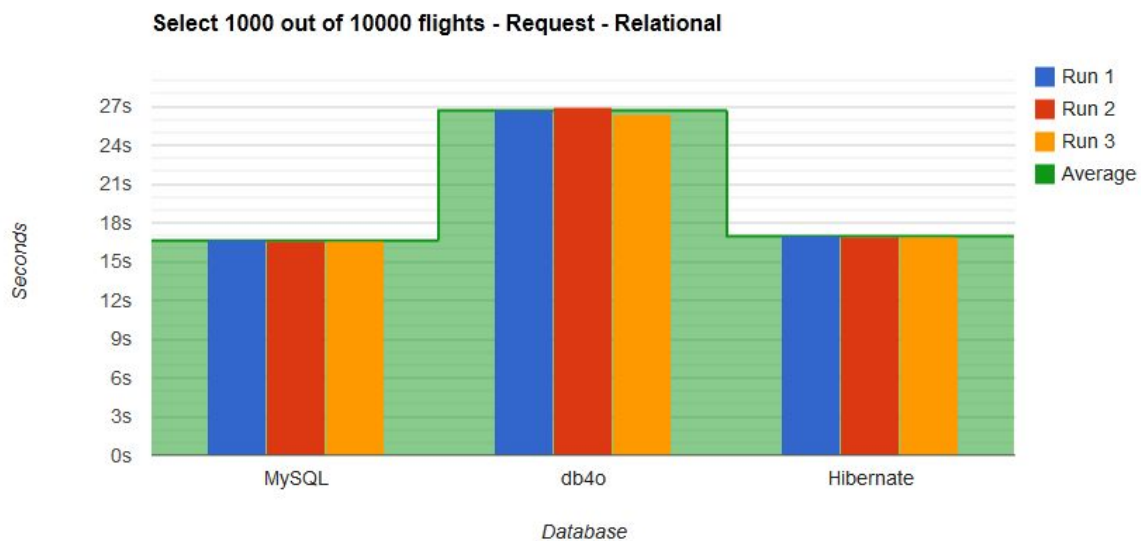


**Figure 104** Total database operation time to do 1000 selects of 1000 flights

	Run 1	Run 2	Run 3	Average	% of request
MySQL	1.893s	1.737s	1.853s	1.827666667s	75.8%
db4o	2.042s	2.109s	2.12s	2.090333333s	84.0%
Hibernate	2.31s	2.296s	2.198s	2.268s	83.8%

**Figure 105** Data behind the chart in *figure 104*

Looking at the chart (*figure 104*) and table (*figure 105*) we can see that there is a larger difference between MySQL and db4o on the database operation level. At the database operation level they differ by about 300 milliseconds. This means that in the test, the non-database operations took longer for MySQL.



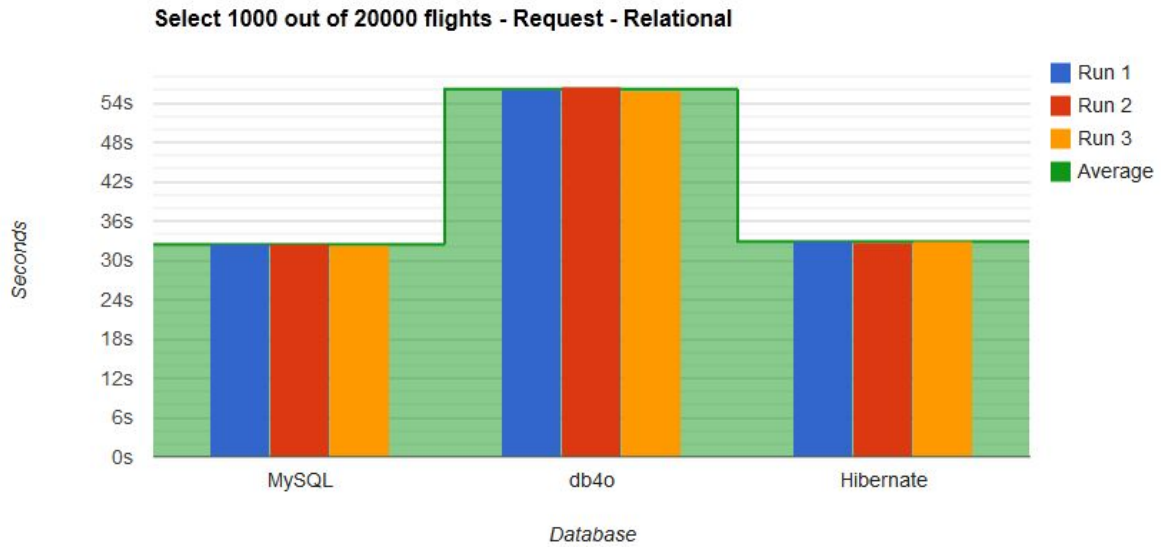
**Figure 106** Total response time to select 1000 flights out of 10 000

	Run 1	Run 2	Run 3	Average	% difference to 1000 flights
MySQL	16.658s	16.631s	16.596s	16.62833333s	+590.1%
db4o	26.738s	26.899s	26.424s	26.687s	+972.3%
Hibernate	17.045s	16.957s	16.907s	16.96966667s	+527.1%

**Figure 107** Data behind the chart in *figure 106*

The dataset has been increased tenfold but it does not mean the total response time is also increased tenfold when it comes to MySQL and Hibernate. MySQL takes 6.9 times longer to search 1000 times in the dataset of 10 000 flights while Hibernate takes 6.3 times longer.

From previously been close to MySQL and Hibernate, db4o loses its closeness when the dataset is increased tenfold. The total response time has seen an increase of 10.7 times the previous result at 1000 flights in the dataset. This suggest that MySQL has some technique to be able to keep up with larger datasets better while db4o has a predictable increase linked to the amount of data.

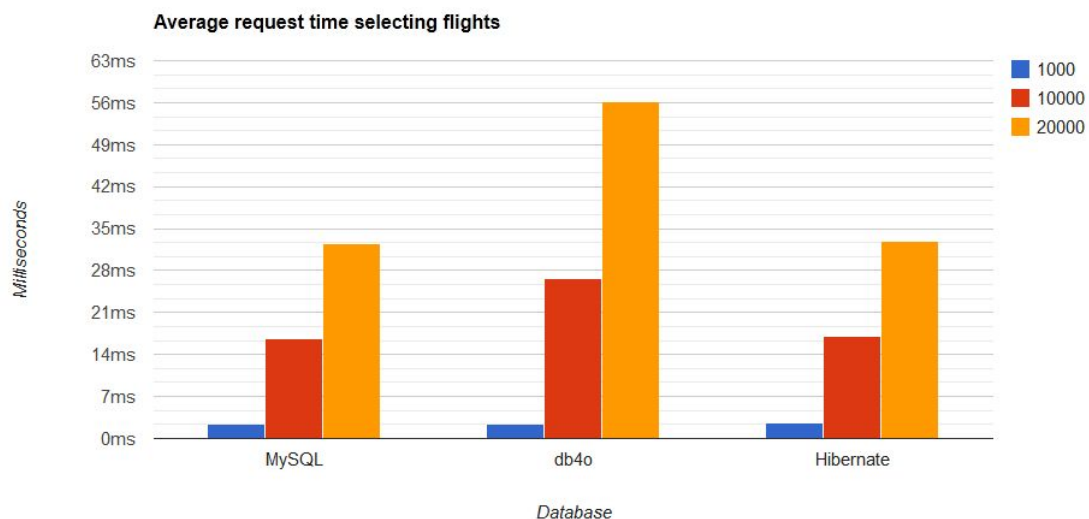


**Figure 108** Total response time to select 1000 flights out of 20 000

	Run 1	Run 2	Run 3	Average	% of difference to 10 000 flights
MySQL	32.535s	32.46s	32.291s	32.42866667s	+95.0%
db4o	56.151s	56.437s	55.773s	56.12033333s	+110.3%
Hibernate	32.887s	32.745s	32.942s	32.858s	+93.6%

**Figure 109** Data behind the chart in *figure 108*

When the dataset has been doubled from 10 000 flights to 20 000 flights, the total response time has almost been doubled too. For MySQL and Hibernate there was an increase of a little bit under twice the time, while for db4o the increase was a bit above twice the time.

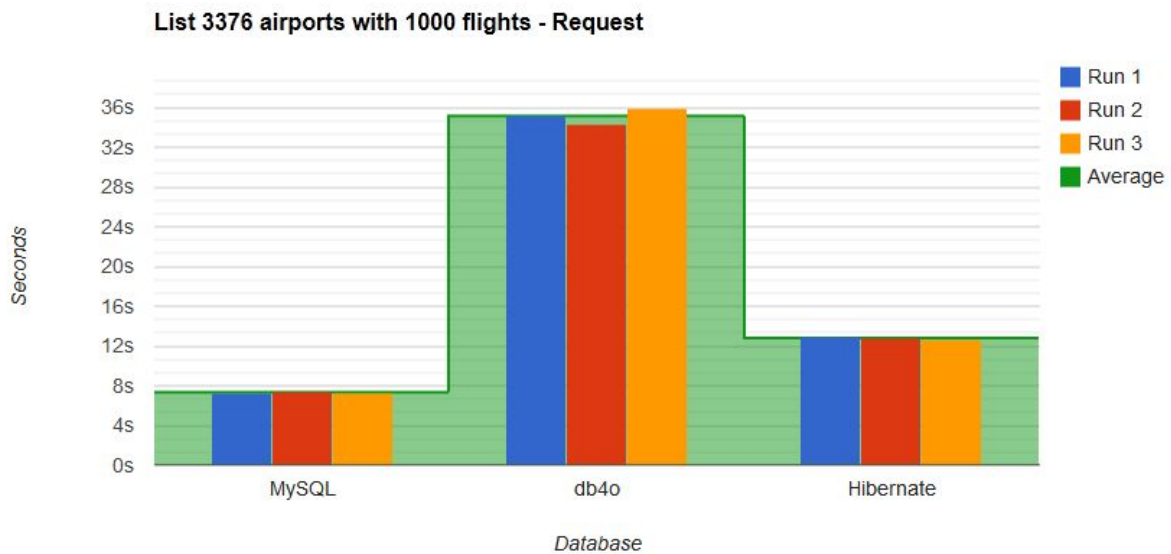


**Figure 110** Average response time selecting flights in each database

	1000	10000	20000
MySQL	2.409666667ms	16.628333333ms	32.428666667ms
db4o	2.488666667ms	26.687ms	56.120333333ms
Hibernate	2.706ms	16.969666667ms	32.858ms

**Figure 111** Data behind the chart in *figure 110*

If each dataset are put next to each other for each database, we can clearly see the increase of the dataset affect the average response time.



**Figure 112** Total response time of listing 3376 airports and their incoming flights, with 1000 flights in the database

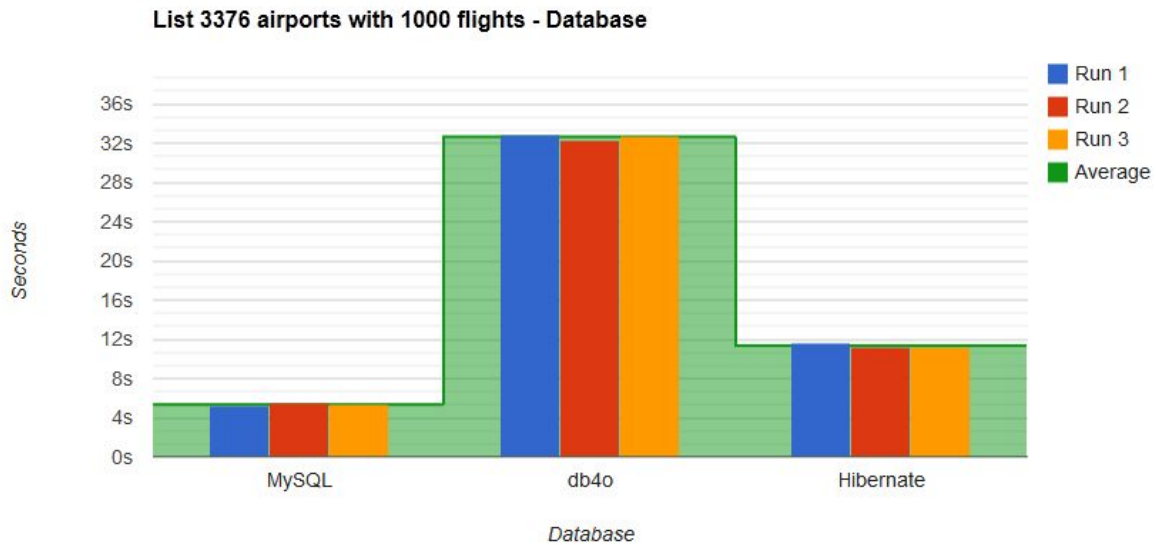
	Run 1	Run 2	Run 3	Average
MySQL	7.317s	7.379s	7.329s	7.341666667s
db4o	35.256s	34.26s	35.943s	35.153s
Hibernate	13.006s	12.753s	12.616s	12.791666667s

**Figure 113** Data behind the chart in *figure 112*

The listing test was meant as a way to find all incoming flights to a provided airport. This was done by fetching the airport, then either fetch all flights with a relationship to that airport by destination (MySQL) or by fetching the list in the **Airport** object (db4o and Hibernate).

Note that in the MySQL (and Hibernate) test, we are searching in an indexed column (unlike the other select tests). This is because we are searching in a column that has a foreign key, and by default, foreign keys are indexed in MySQL when using innodb.

MySQL is once again clearly fastest, followed by Hibernate with a 74 percent overhead. Db4o does not perform any good and continues to be a lot slower than MySQL and Hibernate.

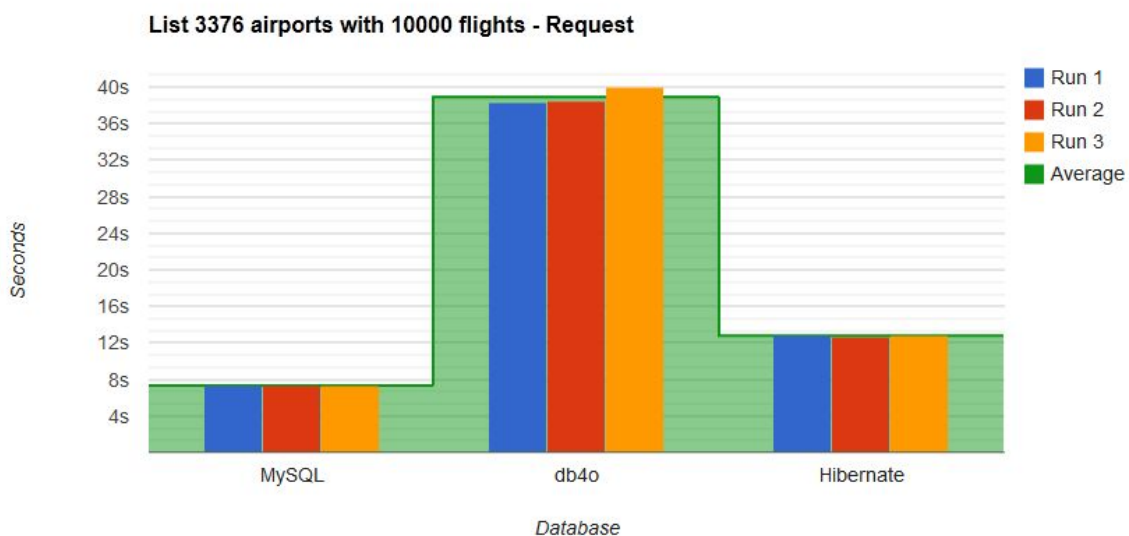


**Figure 114** Total database time of listing 3376 airports and their incoming flights, with 1000 flights in the database

	Run 1	Run 2	Run 3	Average	% of request
MySQL	5.269s	5.457s	5.329s	5.351666667s	72.9%
db4o	32.9s	32.333s	32.704s	32.645666667s	92.9%
Hibernate	11.603s	11.266s	11.206s	11.358333333s	88.8%

**Figure 115** Data behind the chart in figure 114

The database operation times continue to take around the same part of the requests as in previous tests. MySQL is still a winner with Hibernate that adds an overhead of 110 percent on the database operation itself. It seems like the non-database operations have taken a bit more than normal for the MySQL test.



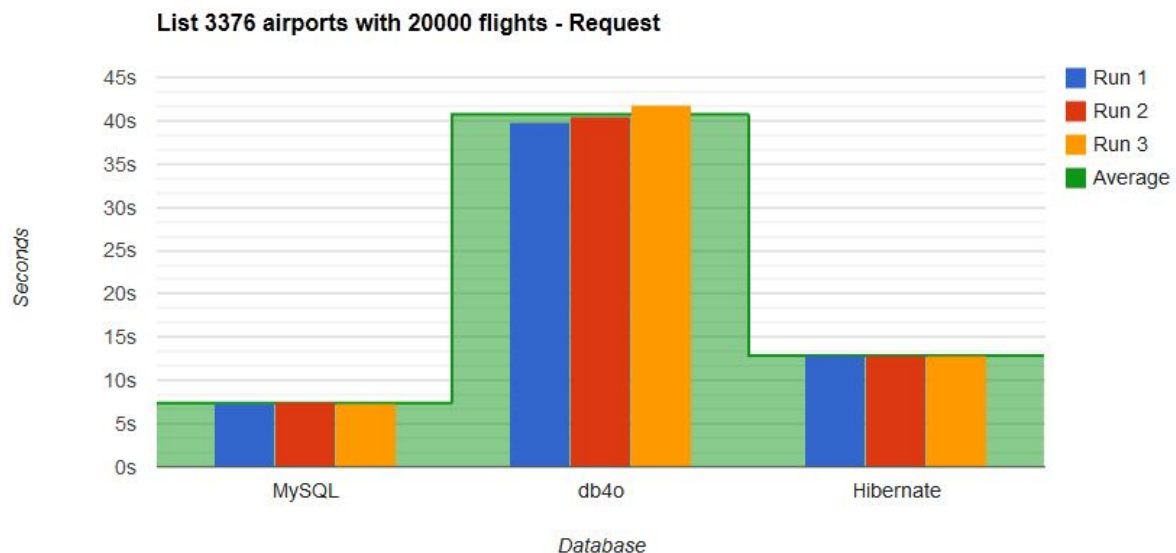
**Figure 116** Total response time of listing 3376 airports and their incoming flights, with 10 000 flights in the database

	Run 1	Run 2	Run 3	Average	% difference to 1000 flights
MySQL	7.369s	7.328s	7.318s	7.338333333s	-4.5%
db4o	38.281s	38.529s	39.975s	38.92833333s	+10.7%
Hibernate	12.867s	12.718s	12.761s	12.782s	-7.6%

**Figure 117** Data behind the chart in *figure 116*

Interestingly, even though the amount of flights have been increased tenfold, MySQL and Hibernate performs faster than in the test with only 1000 flights in the dataset. As previously mentioned, the column being searched in the flights table (in MySQL, and Hibernate that uses MySQL as backend) is indexed. Indexing a column makes it faster to search in, even in larger datasets. It could possibly lead to about the same response time in both datasets, and the difference can be attributed to small response variances in the request and database operations.

Db4o, however, does not use any kind of index, yet the total response time only increases 1.1 times. The airport object contains a list of all incoming flights, therefore it does not need to take an id from an airport and do another search in the flight storage, as MySQL and Hibernate needs to do. When fetching the airport, the flights are fetched as well.

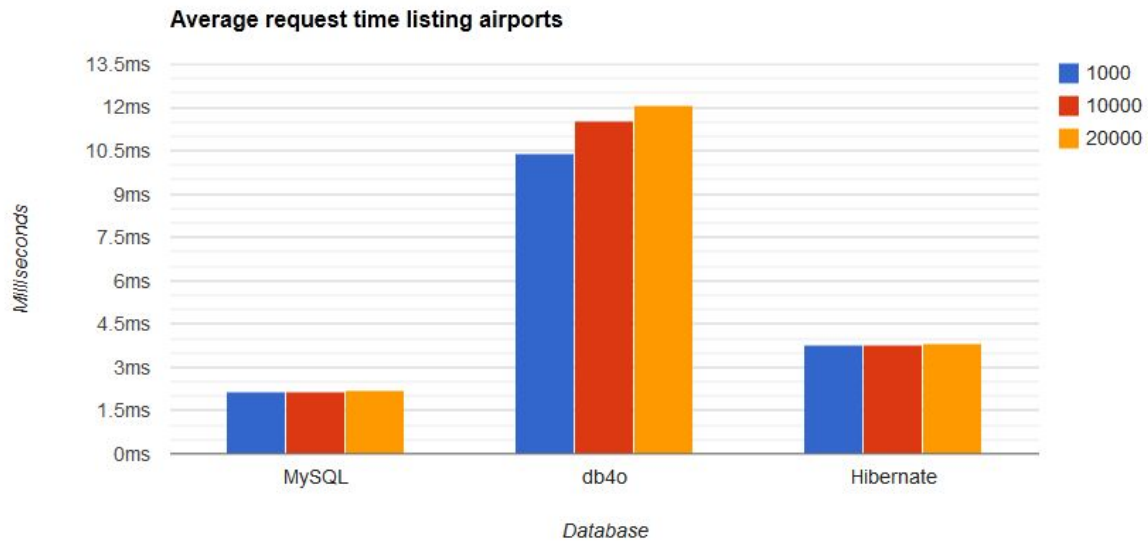


**Figure 118** Total response time of listing 3376 airports and their incoming flights, with 20 000 flights in the database

	Run 1	Run 2	Run 3	Average	% difference to 10 000 flights
MySQL	7.344s	7.463s	7.395s	7.400666667s	+0.8%
db4o	39.878s	40.546s	41.742s	40.722s	+4.6%
Hibernate	12.828s	12.936s	12.894s	12.886s	+0.8%

**Figure 119** Data behind the chart in *figure 118*

When the flight dataset has been doubled, the response time is still not affected for MySQL and Hibernate. The total response time for db4o did not change very much either, with an increase of 1.8 seconds.



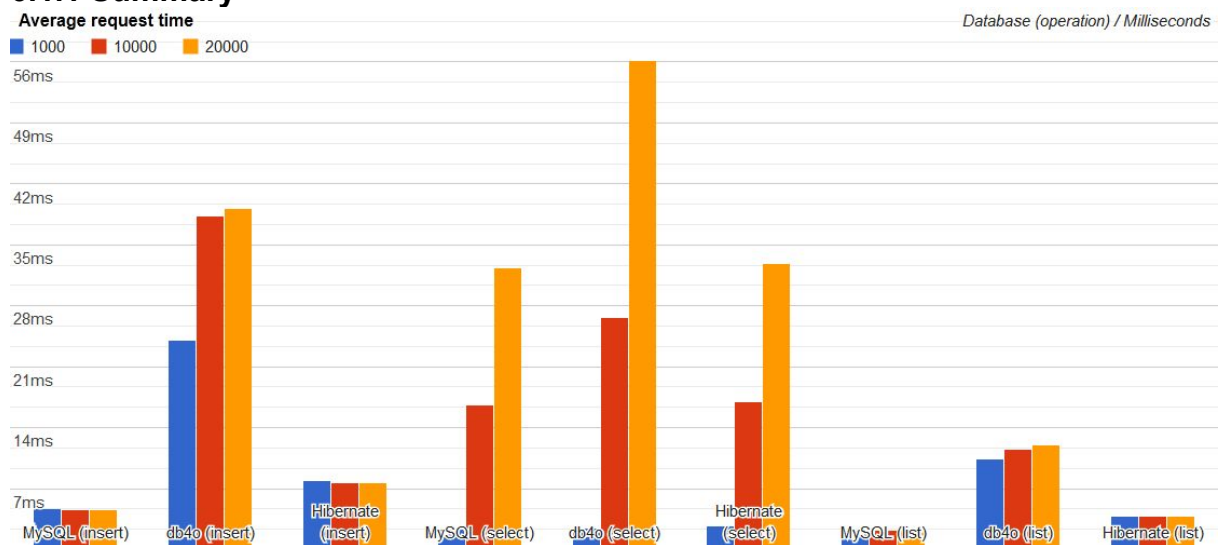
**Figure 120** Average request time to list airports with different amounts of flights

	1000	10000	20000
MySQL	2.174664297ms	2.173676935ms	2.1921406ms
db4o	10.41261848ms	11.53090442ms	12.06220379ms
Hibernate	3.78900079ms	3.786137441ms	3.816943128ms

**Figure 121** Data behind the chart in figure 120

Looking at the average request time for each database and number of flights, MySQL and Hibernate seems to be unchanged, at an average of 2.2 milliseconds and 3.8 milliseconds respectively. Db4o however seem to increase a little as the number of flights increases.

## 6.1.1 Summary



**Figure 122** Summary of the average time of each response on each database



	1000	10000	20000
MySQL (insert)	4.657ms	4.589533333ms	4.578566667ms
db4o (insert)	24.08466667ms	38.3782ms	39.22525ms
Hibernate (insert)	7.942333333ms	7.717066667ms	7.71625ms
MySQL (select)	2.409666667ms	16.62833333ms	32.42866667ms
db4o (select)	2.488666667ms	26.687ms	56.12033333ms
Hibernate (select)	2.706ms	16.96966667ms	32.858ms
MySQL (list)	2.174664297ms	2.173676935ms	2.1921406ms
db4o (list)	10.41261848ms	11.53090442ms	12.06220379ms
Hibernate (list)	3.78900079ms	3.786137441ms	3.816943128ms

**Figure 123** Data behind the chart in *figure 122*

*Figure 122* shows a quick summary of average response time for each operation and for each database. It includes the average for inserting flights, selecting flights and retrieving all incoming flights to an airport. More detailed charts can be found in *figure 98*, *figure 110* and *figure 120*.

## 6.2 Analysis

There are lots of differences between each software and how they perform in different situations. While performing very similar to the rest in one test, it can perform very different in another test.

When testing the most basic things like inserting simple objects with no relations to another object or sets of multiple values, both databases and the ORM framework performed very similarly. As the dataset increased, so did the total response time, at roughly the same rate for both databases and the ORM framework. With an average within 0.6 milliseconds per request of each other, the choice of database might depend on if you are ready to sacrifice those milliseconds when inserting data. After all, would you insert 100 000 records into the database, you would only lose one minute if you pick Hibernate, if going by the insert average response time where Hibernate had the largest difference.

Searching in the database is another story. Db4o performed better than MySQL and Hibernate when searching in a database of 1000 flights. However, when the dataset was increased tenfold, db4o's total response time increased tenfold as well. That might sound reasonable considering it ought to happen when you increase the dataset tenfold, but MySQL and Hibernate did not increase tenfold, increasing 6.8 and 6.2 times instead, respectively. That means that db4o suddenly performed worse than MySQL and Hibernate. It seems like db4o has a fairly linear increase in its response time as the data is increased, while MySQL (and Hibernate, since it uses MySQL as a back-end) does not increase the same way.

Making the objects more complex affects the response time a lot. Airport objects were introduced, and the field in the Flight object that was previously represented by a string is now represented by this object. A table was created in the relational database to hold these airports, and two foreign keys were created between the airports and flights table, linking airport id's to the flights origin and destination.

Inserting 3376 airports were no problem for any of the databases, since it is a simple object, much

like flight objects in the non-relational version. MySQL was the fastest database to insert all airports, followed by Hibernate that was roughly one second slower. Db4o was two seconds slower than MySQL.

Inserting flights, however, showed the true performance of db4o. While MySQL inserted all 1000 flights in 4.7 seconds on average and Hibernate in 7.9 seconds on average, db4o took 24 seconds to insert all 1000 flights. Since the flight objects received from the client contains airport objects where only the IATA code was set, the proper airports were required to be fetched before the flight could be inserted, which added some time onto each database. Why db4o took five times as long as MySQL is unknown and further testing needs to be done to figure out the cause. The most likely theory is that db4o is not very efficient at storing lists and sets, since the airport object contains a list of incoming and departing flights, and these have to be updated each time a flight is processed.

In terms of when trying to insert 10 000 flights, db4o became very inconsistent with its performance. From taking 359 seconds to insert 10 000 flights in the first run, it went to 393 seconds on the second run. Even when the test was retaken, different results were achieved. At 20 000 flights it was even more inconsistent, with values ranging between 592 seconds and 962 seconds, reaching up to 1100 seconds in test runs.

It is unclear why the results were so varied, even though it was the same test that was being run. Some theories have been in regards of memory usage and garbage collection but it is uncertain if that is the cause. Another theory is that something temporary would occur in the operating system just at that point, for example automatic maintenance, but it was disproved when retaking the test several times, getting the same result. To get a clearer idea why db4o performs like this, we would have to do further testing specifically to db4o, which would be out of scope for this work.

Selecting flights in the databases were quite straightforward. At a smaller dataset of 1000 flights, MySQL was the fastest. Db4o came in second, slower by 200 milliseconds than MySQL, and Hibernate came last by roughly 400 milliseconds slower than MySQL.

In the dataset of 10 000 flights, db4o had a bit more than a tenfold increase, like the increase in the non-relational version. However, MySQL and Hibernate did not increase by the same amount. That made db4o ten seconds slower than MySQL, while Hibernate was only three seconds slower than MySQL.

Doubling the dataset to 20 000 flight, the total response time of db4o was increased by 2.1 times the total response time of inserting 10 000 flights, while MySQL and Hibernate had roughly an increase by two times the previous dataset. While MySQL selected 1000 flights out of 20 000 in a total of 32.4 seconds on average, Hibernate did the same thing in 32.8 seconds and db4o in 56.1 seconds. It was observed that the more data there is, the less overhead did Hibernate seem to have to MySQL, but more tests at larger datasets would need to be done to ensure that is the case.

The last test performed was retrieving an airport and fetching all incoming flights to that airport. All 3376 airports were retrieved, with different amounts of flights in the database. As mentioned before, in this test selects were made on indexed columns in MySQL and Hibernate, since MySQL automatically indexes foreign keys by default. An airport was specified by IATA. In the MySQL test, this meant the airport had to be retrieved first to get its id, then another select would need to be done on the flights table, searching for that id in the Dest column. Its probable that Hibernate does a similar approach, selecting the airports first, then selecting the flights. Db4o however did not have an index, but the airport object contained a list of incoming flights. Therefore, when fetching an airport, the list with flights was fetched as well.

The amount of flights did not seem to matter to MySQL and Hibernate at all. In 1000, 10 000 and 20 000 flights, MySQL fetched everything in about 7.4 seconds while hibernate fetched everything in about 12.8 seconds. It is probable that the indexed foreign key increases the search time that there is no big difference in searching in 1000, 10 000 and 20 000 flights. A larger dataset would be needed to see where one start to see a difference.

Db4o, however, showed a small increase in total response time as the dataset was increased. At 1000 flights, db4o took 35 seconds to retrieve everything needed, at 10 000 flights it increased to 39 seconds, and at 20 000 flights it reached 40.7 seconds on average.

### **6.3 Conclusion**

Our conclusion is that MySQL and Hibernate had similar results, yet as expected, Hibernate does add an overhead to the database operation and response time compared to pure MySQL. Db4o is mostly out of the question based on how poorly it performed in regards of response time. Db4o could be acceptable in local applications where you do not work with large amounts of data or large amounts of requests in a short period, where response time is not important. In a web environment it is doomed to fail in its current state, since web pages are expected to load fast, and as traffic increases and the number of requests increases, the web pages should still load within a reasonable period.

Hibernate was slower than MySQL by about 90% when inserting data, but was very close to MySQL when searching data in the database. If your application will mostly retrieve data and not insert data very often at high rates, Hibernate might be an option if one would find it easier and more comfortable to work with instead of pure SQL in the code.

All databases and the ORM framework has been used out-of-the-box. No performance optimisations have been attempted, and all code and queries are based on “getting started” guides from the respective softwares documentation or tutorials on other sites. There is a possibility that if we had more time to delve deeper into the documentations and spend more time optimizing the databases and the ORM framework, we might have gotten a better result, especially for db4o. Further studies have to be made where optimisation are the focus to be able to make certain how each database behaves in its most optimal setting.

## 7 Concluding Remarks

### 7.1 Summary

There are several types of databases with different features. There are also tools that help you use these databases in a different way than intended, a way that may be easier and more logical in the context of the application. In this work we explored the response time of the relational database MySQL and compared it to the object database db4o. We also explored the performance of the ORM framework Hibernate, which could be seen as a way to use a relational database like an object database. We wanted to see how this affects a web application, so a web application was made to test the different tools. It was found that MySQL performs the fastest on every test, while Hibernate came close second, and db4o performed worst on almost all test except for a few.

### 7.2 Discussion

It was not a big surprise that MySQL would perform the best in most tests based on prior articles discussing the topic and from prior experience with some of the tools. What was surprising was the insert tests of db4o and MySQL. The study by Roopak et al. (2013) claimed that db4o would win over MySQL by far when it came to inserting data. 20 000 simple objects would take 2.767 seconds to be inserted in db4o and 8.685 seconds to insert the data into MySQL. However, in our non-relational test which was identical to what Roopak et al. (2013) did, our results were that it took 25.96 seconds to insert 20 000 objects into the db4o database and 19.474 seconds to insert 20 000 rows into the MySQL database.

Roopak et al. (2013) did not mention which hardware or configuration they were using. They did not disclose the code behind the test, neither which version of the softwares they were using. It might be reasonable that they had better hardware than the one used in this work, however it does not explain why we found db4o to be slower at inserting data than MySQL, when Roopak et al. found the opposite. They claimed that db4o was slower than MySQL to search data in the database, which we found to be mostly true. At the dataset of 1000 flights, db4o actually had less response time than MySQL, however at 10 000 and 20 000 flights, db4o had a considerably higher response time.

Van Zyl et al. (2006) did find db4o to be faster than Hibernate when performing the OO7 database test, where Hibernate took 2.5 times longer to insert data than db4o. Although the data used in our work is different from the data used in van Zyl et al. (2006), we can see that it is probable that our configuration and use of db4o may not have been very optimal. Van Zyl et al. (2006) does not mention any specific configuration other than changing the default parameter of update depth when storing objects that contain sets of references to other objects. This should not have any effect when storing simple objects with no references to other objects, but it has not been tested.

Some of the results have been hard to explain and no proper conclusion have been made on these results. For example, considering the inconsistent results when inserting large amounts of flights into db4o. There would need to be further testing and investigation to find out the cause of these inconsistent results is. Some solutions have been tried like increasing the amount of heap space allocated to the Java virtual machine, but it has not had any effect.

We believe most of our results are as correct they can be given the circumstances of this work. We feel like more tests needs to be done to fully ensure as accurate results as possible. We have tried to do whatever we can to make sure no outside forces are affecting the results. For example, the time between the computers running the server and the client have been synced locally using NTP. All processes and applications that are not needed have been terminated prior to each test, and we have made sure automatic maintenance on the computers have been disabled or set to a time

when we did not run any tests.

However, we cannot fully ensure that the results have not been affected negatively from outside forces. Since the tests take a very long time to run and the computers running the tests being completely unusable when running the tests, all tests have been run unattended over the night. We can not be certain that the operating systems did not do any background tasks unrelated to the tests. Supervising the test would not solve that issue since it is hard to detect what the system is doing at any given moment. A way to solve that was to run the same tests several times at different times, which we did and got similar results every time.

We have made sure to release all our code, both attached to this work and uploaded publically to GitHub for easier download. We have also made sure to disclose which hardware that have been used, what versions of the software we have used and how they were configured. We have also described how the tests work and how we ran them, and what precautions have been made to make as accurate tests as possible. We believe this is enough for others to recreate our tests and even build upon them, going further with optimisations, different tests and so on.

No personal information have been collected whatsoever during the study. All results that were collected are purely of timestamps of each request as well as some basic info about which test and database were used. Therefore there are no ethical issues regarding personal details or any other sensitive details. There are no cultural aspects since no focus have been made other than on the specific results of the software chosen.

The results we found can be used as a way to get a perspective about the performance of the databases that were tested and the ORM framework that was tested regarding response time. This can make it easier to pick the right tool for the job when in the planning stage of software development. However, this study alone would not be enough in itself to be able to make a proper choice. There are other aspects to be considered that was not brought up in this study, like development time, cost of the product, license, support and size as well as other things. To be able to make a proper choice, one is suggested to compare different studies, compare features of each database and figure out what the application being developed needs.

We believe the study can show a quick overview of the software used, but further testing needs to be done when deciding what software to use in the project.

This work made three additions to the study done by Roopak et al. (2013). The largest being the addition of a web environment. The second addition that was done was the addition of the Hibernate ORM framework, where we performed the same (and some additional) tests as Roopak et al. (2013) did on the object database and relational database. Lastly, we introduced a table of airports to create a more complex data structure with relations to see how each database fares with different data structures.

### **7.3 Future work**

With a couple of more weeks, we could have run all the tests several times more and try to figure out what is causing some of the tests to return unexpected results. Different tweaks could have been made, like changing basic settings and configurations of the databases and the ORM framework. The tests could have been extended to other computers with different hardware and operating systems as well in an attempt to see how the results differ between the computers.

If we had several months more, we could delve deep into the documentation of each database to learn about all different kind of tweaks, notes and optimisations. That way we could have tested each database and the ORM framework in their most optimal environment, instead of their

out-of-the-box environment.

A future work could extend this work by focusing on optimisation. There is probably a lot of things one could do to make the software run better, as no consideration to optimisation was made during this study. One example is to test indexing the tables and searching in the columns that have been indexed and compare it to the non-indexed columns. One could explore how indexes in db4o affects the performance of db4o (db4o, n.db).

Another thing not tested in this work that would be interesting to test if there would have been more time is inheritance, since it is a big part of object-oriented programming.

We focused on relational and object-oriented databases in this work. Had there been more time then we could test other types of databases as well—for example—document databases and object-relational databases.

Some efforts were made to explain and make a conclusion about the inconsistent results of the db4o insert tests in the relational version, but none had any effect. With more time, we could have investigated further and tried to understand what the cause is.

When in the planning phase of software development it is important to pick the right tools for the job. This study might be useful to see some basic tests of response time of different databases, but may lack some crucial parts. A company or developer could take this study and build upon it to fit their applications needs, implementing tests close to what their application will do, and run their own tests to get a more specific and useful view for their application.

## References

- American Statistical Association. (2009a) The data. Data expo 09. ASA Statistics Computing and Graphics. Available at Internet: <http://stat-computing.org/dataexpo/2009/the-data.html> [Retrieved March 11, 2014].
- American Statistical Association. (2009b) Supplemental data. Data expo 09. ASA Statistics Computing and Graphics. Available at Internet: <http://stat-computing.org/dataexpo/2009/supplemental-data.html> [Retrieved March 11, 2014].
- Bagui, S., 2003. Achievements and Weaknesses of Object-Oriented Databases. *Journal of Object Technology* 2, 29–41.
- Bazghandi, A., 2006. Web Database Connectivity Methods (using Mysql) in Windows Platform, in: *Information and Communication Technologies, 2006. ICTTA '06. 2nd. Presented at the Information and Communication Technologies, 2006. ICTTA '06. 2nd*, pp. 3577–3581.
- db4o. (n.da) *db4o :: Java & .NET Object Database – Open Source Object Database, Open Source Persistence, Oodb*. Available at Internet: <http://db4o.com/> [Retrieved April 17, 2014].
- db4o. (n.db) *db4o tutorial*. Available at Internet: <http://community.versant.com/documentation/reference/db4o-8.0/java/tutorial/> [Retrieved April 17, 2014].
- Di Giacomo, M., 2005. MySQL: lessons learned on a digital library. *IEEE Software* 22, 10–13.
- Effective MySQL. (2011) *Primary key -- Effective MySQL*. Available at Internet: <http://effectivemysql.com/article/primary-key/> [Retrieved April 14, 2014].
- Gao, T. (2012). *Tricks for Better Software: Using db4o in Scala Programs*. Available at Internet: <http://ted-gao.blogspot.se/2012/12/using-db4o-in-scala-programs.html> [Retrieved March 25, 2014].
- Ghosh, P., Rau-Chaplin, A., 2006. Performance of Dynamic Web Page Generation for Database-driven Web Sites, in: *International Conference on Next Generation Web Services Practices, 2006. NWeSP 2006. Presented at the International Conference on Next Generation Web Services Practices, 2006. NWeSP 2006*, pp. 56–63.
- Goschka, K.M., Riedling, E., 1997. Development of an object oriented framework for design and implementation of database powered distributed Web applications with the DEMETER project as a real-life example, in: *EUROMICRO 97. "New Frontiers of Information Technology". Short Contributions., Proceedings of the 23rd Euromicro Conference. Presented at the EUROMICRO 97. "New Frontiers of Information Technology". Short Contributions., Proceedings of the 23rd Euromicro Conference*, pp. 132–137.
- Hibernate. (n.da) *Hibernate ORM - Hibernate ORM*. Available at Internet <http://hibernate.org/orm/> [Retrieved April 15, 2014].
- Hibernate. (n.db) *Chapter 2. Tutorial Using Native Hibernate APIs and hbm.xml Mappings*. Available at Internet: <http://docs.jboss.org/hibernate/orm/4.2/quickstart/en-US/html/ch02.html> [Retrieved April 16, 2014].
- Ireland, C., Bowers, D., Newton, M., Waugh, K., 2009. A Classification of Object-Relational Impedance Mismatch, in: *First International Conference on Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. Presented at the First International*

- Conference on Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09, pp. 36–43.
- jOOQ. (2013) *10 More Common Mistakes Java Developers Make when Writing SQL | Java, SQL and jOOQ*. Available at Internet: <http://blog.jooq.org/2013/08/12/10-more-common-mistakes-java-developers-make-when-writing-sql/> [Retrieved April 15, 2014].
- Kienzle, J., Romanovsky, A., 2002. Framework based on design patterns for providing persistence in object-oriented programming languages. *Software, IEE Proceedings* - 149, 77–85.
- Lange, D.B., 1994. An abstract model of the object-oriented DBMS, in: *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, 1994. Presented at the *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, 1994, pp. 807–816.
- Liu, H., Hong, Y., Hao, H., Wang, C., 2008. Kernel: A RDB-Based Object Persistence Component Set for Java, in: *2008 International Conference on Computer Science and Software Engineering*. Presented at the *2008 International Conference on Computer Science and Software Engineering*, pp. 64–67.
- Mann, R.S., Devgan, S.S., 2000. Implementation of embedded streaming of large video application using object-relational database and PHP, in: *Proceedings of the IEEE Southeastcon 2000*. Presented at the *Proceedings of the IEEE Southeastcon 2000*, pp. 201–204.
- Mkyong. (2013) *How to send HTTP request GET/POST in Java*. Available at Internet: <http://www.mkyong.com/java/how-to-send-http-request-getpost-in-java/> [Retrieved April 19, 2014].
- Ohara, M., Nagpurkar, P., Ueda, Y., Ishizaki, K., 2009. The data-centricity of Web 2.0 workloads and its impact on server performance, in: *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. ISPASS 2009. Presented at the *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. ISPASS 2009, pp. 133–142.
- Patel, V. (2011) *Your first Play! – GAE – Siena application: Tutorial with Example*. Available at Internet: <http://viralpatel.net/blogs/first-play-framework-gae-siena-application-tutorial-example/> [Retrieved April 18, 2014].
- Play Framework. (n.da) *Play Framework - Build Modern & Scalable Web Apps with Java and Scala*. Available at Internet: <http://www.playframework.com/> [Retrieved March 25, 2014].
- Play Framework. (n.db) *JavaToDoList*. Available at Internet: <http://www.playframework.com/documentation/2.2.x/JavaToDoList> [Retrieved March 25, 2014].
- Play Framework. (n.dc) *Production*. Available at Internet: <http://www.playframework.com/documentation/2.2.2/Production> [Retrieved May6, 2014].
- Prajapati, H.B., Dabhi, V.K., 2009. High Quality Web-Application Development on Java EE Platform, in: *Advance Computing Conference*, 2009. IACC 2009. IEEE International. Presented at the *Advance Computing Conference*, 2009. IACC 2009. IEEE International, pp. 1664–1669.
- Quigley, E., Gargenta, M., 2006. *PHP and MySQL by Example*. Prentice Hall.
- Roopak, K.E., Rao, K.S.S., Ritesh, S., Chickerur, S., 2013. Performance Comparison of Relational Database with Object Database (DB4o), in: *2013 5th International Conference on*



- Computational Intelligence and Communication Networks (CICN). Presented at the 2013 5th International Conference on Computational Intelligence and Communication Networks (CICN), pp. 512–515.
- Schahczenski, C., 2000. Object-oriented Databases in Our Curricula, in: Proceedings of the Seventh Annual CCSC Midwestern Conference on Small Colleges. Consortium for Computing Sciences in Colleges, USA, pp. 170–176.
- Stackoverflow. (2012) *MySQL - Persistent connection vs connection pooling*. Available at Internet: <http://stackoverflow.com/questions/9736188/mysql-persistent-connection-vs-connection-pooling> [Retrieved April 19, 2014].
- Tutorialspoint. (n.da) *Hibernate Configuration*. Available at Internet: [http://www.tutorialspoint.com/hibernate/hibernate\\_configuration.htm](http://www.tutorialspoint.com/hibernate/hibernate_configuration.htm) [Retrieved April 15, 2014].
- Tutorialspoint. (n.db) *Hibernate Mapping Files*. Available at Internet: [http://www.tutorialspoint.com/hibernate/hibernate\\_mapping\\_files.htm](http://www.tutorialspoint.com/hibernate/hibernate_mapping_files.htm) [Retrieved April 16, 2014].
- Tutorialspoint. (n.dc) *Hibernate Annotations*. Available at Internet: [http://www.tutorialspoint.com/hibernate/hibernate\\_annotations.htm](http://www.tutorialspoint.com/hibernate/hibernate_annotations.htm) [Retrieved April 16, 2014].
- Tutorialspoint. (n.dd) *Hibernate Many-to-One Mappings*. Available at Internet: [http://www.tutorialspoint.com/hibernate/hibernate\\_many\\_to\\_one\\_mapping.htm](http://www.tutorialspoint.com/hibernate/hibernate_many_to_one_mapping.htm) [Retrieved April 16, 2014].
- Tutorialspoint. (n.de) *Hibernate One-to-Many Mappings*. Available at Internet: [http://www.tutorialspoint.com/hibernate/hibernate\\_one\\_to\\_many\\_mapping.htm](http://www.tutorialspoint.com/hibernate/hibernate_one_to_many_mapping.htm) [Retrieved April 16, 2014].
- Tutorialspoint. (n.df) *Hibernate Query Language*. Available at Internet: [http://www.tutorialspoint.com/hibernate/hibernate\\_query\\_language.htm](http://www.tutorialspoint.com/hibernate/hibernate_query_language.htm) [Retrieved April 17, 2014].
- Van Zyl, P., Kourie, D.G., Boake, A., 2006. Comparing the Performance of Object Databases and ORM Tools, in: Proceedings of the 2006 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries, SAICSIT '06. South African Institute for Computer Scientists and Information Technologists, Republic of South Africa, pp. 1–11.
- Van Zyl, P., Kourie, D.G., Coetzee, L., Boake, A., 2009. The Influence of Optimisations on the Performance of an Object Relational Mapping Tool, in: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT '09. ACM, New York, NY, USA, pp. 150–159.
- Walek, B., Bartos, J., Klimes, C., 2012. Fuzzy tool for reasoning in object relational mapping of information system, in: 2012 35th International Conference on Telecommunications and Signal Processing (TSP). Presented at the 2012 35th International Conference on Telecommunications and Signal Processing (TSP), pp. 81–85.
- Wegrzynowicz, P., 2013. Performance antipatterns of one to many association in hibernate, in: 2013 Federated Conference on Computer Science and Information Systems (FedCSIS). Presented at the 2013 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 1475–1481.
- Winand, M. (2012a). *Improving SQL Join Performance by Indexing | SQL Performance Explained*. Available at Internet: <http://use-the-index-luke.com/sql/join> [Retrieved April 15, 2014].

Winand, M. (2012b). *Anatomy of an SQL Index: What is an SQL Index | SQL Performance Explained*. Available at Internet <http://use-the-index-luke.com/sql/anatomy> [Retrieved April 15, 2014].

Windl, U., Dalton, D., Martinec, M., Worley, D.R. (2006) *The NTP FAQ and HOWTO*. Available at Internet: <http://www.ntp.org/ntpfaq/> [Retrieved May 7, 2014].

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. *Experimentation in Software Engineering*. Springer-Verlag, Berlin Heidelberg.

# Appendices

To ease replication of my tests all relevant code needed to run my tests (except for the play framework itself and maven repositories), all code is available for download at <https://github.com/galaxyAbstractor/thesis> for the non-relational version and <https://github.com/galaxyAbstractor/thesis/tree/relational> for the relational version.

## Appendix 1

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {
    @Id @GeneratedValue
    @Column(name = "id")
    private int id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "salary")
    private int salary;

    public Employee() {}
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

(Tutorialspoint, n.dc)

## Appendix 2

```
import java.util.*;

public class Employee{
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
    private Address address;

    public Employee() {}
    public Employee(String fname, String lname,
                    int salary, Address address ) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
        this.address = address;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress( Address address ) {
        this.address = address;
    }
}
```

(Tutorialspoint, n.dd)

## Appendix 3

```
import java.util.*;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
    private Set certificates;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }

    public Set getCertificates() {
        return certificates;
    }
    public void setCertificates( Set certificates ) {
        this.certificates = certificates;
    }
}
```

(Tutorialspoint, n.de)

## Appendix 4

File: \$root/server/conf/routes

```
POST    /insert/airport/one/:time      controllers.Insert.oneAirport(time: Long)
POST    /insert/flight/one/:time      controllers.Insert.oneFlight(time: Long)

GET      /select/flight/deptime/:depTime/:time
controllers.Select.flightByDepTime(depTime: Int, time: Long)
GET      /select/join/flight/dest/:dest/:time
controllers.Select.joinFlightByDest(dest: String, time: Long)

GET      /purge
controllers.Purge.purge()
```

## Appendix 5

```
File: $root/server/conf/application.conf
# This is the main configuration file for the application.
# ~~~~~

# Secret key
# ~~~~~
# The secret key is used to secure cryptographics functions.
# If you deploy your application to several instances be sure to use the same key!
application.secret="pXs4o^V<ogVfpVGUS;gre1piK/k:WrXeNwc0Id/sqs7hQs19=`ieu7AToGpS0c;q"

# The application languages
# ~~~~~
application.langs="en"

# Global object class
# ~~~~~
# Define the Global object class for this application.
# Default to Global in the root package.
application.global=Global

# Router
# ~~~~~
# Define the Router object to use for this application.
# This router will be looked up first when the application is starting up,
# so make sure this is the entry point.
# Furthermore, it's assumed your route file is named properly.
# So for an application router like `conf/my.application.Router`,
# you may need to define a router file `my.application.routes`.
# Default to Routes in the root package (and `conf/routes`)
application.router=my.application.Routes

# Database configuration
# ~~~~~
# You can declare as many datasources as you want.
# By convention, the default datasource is named `default`
#
db.default.driver=com.mysql.jdbc.Driver

db.default.url="jdbc:mysql://localhost:3306/testdb"
db.default.user=root
db.default.password=""

# In order to reduce lock contention and thus improve performance,
# each incoming connection request picks off a connection from a
# pool that has thread-affinity.
# The higher this number, the better your performance will be for the
# case when you have plenty of short-lived threads.
# Beyond a certain threshold, maintenance of these pools will start
# to have a negative effect on performance (and only for the case
# when connections on a partition start running out).
db.default.partitionCount=5

# The number of connections to create per partition. Setting this to
# 5 with 3 partitions means you will have 15 unique connections to the
# database. Note that BoneCP will not create all these connections in
# one go but rather start off with minConnectionsPerPartition and
# gradually increase connections as required.
db.default.maxConnectionsPerPartition=30

# The number of initial connections, per partition.
db.default.minConnectionsPerPartition=30
```

```

# When the available connections are about to run out, BoneCP will
# dynamically create new ones in batches. This property controls
# how many new connections to create in one go (up to a maximum of
# maxConnectionsPerPartition). Note: This is a per-partition setting.
db.default.acquireIncrement=1

# After attempting to acquire a connection and failing, try to
# connect this number of times before giving up.
db.default.acquireRetryAttempts=10

# How long to wait before attempting to obtain a
# connection again after a failure.
db.default.acquireRetryDelay=5 seconds

# The maximum time to wait before a call
# to getConnection is timed out.
db.default.connectionTimeout=1 second

# Idle max age
db.default.idleMaxAge=10 minute

# This sets the time for a connection to remain idle before sending a test query to the DB.
# This is useful to prevent a DB from timing out connections on its end.
db.default.idleConnectionTestPeriod=5 minutes


#
# You can expose this datasource via JNDI if needed (Useful for JPA)
db.default.jndiName=DefaultDS

# Evolutions
# ~~~~~
# You can disable evolutions if needed
# evolutionplugin=disabled

# Ebean configuration
# ~~~~~
# You can declare as many Ebean servers as you want.
# By convention, the default server is named `default`
#
# ebean.default="models.*"

# Logger
# ~~~~~
# You can also configure logback (http://logback.qos.ch/),
# by providing an application-logger.xml file in the conf directory.

# Root logger:
logger.root=ERROR

# Logger used by the framework:
logger.play=INFO

# Logger provided to your application:
logger.application=DEBUG

```



## Appendix 6

File: \$root/server/build.sbt

```
name := "dbtest"
```

```
version := "1.0-SNAPSHOT"
```

```
libraryDependencies += Seq(  
  javaJdbc,  
  javaEbean,  
  cache,  
  "com.google.code.gson" % "gson" % "2.2.4",  
  "mysql" % "mysql-connector-java" % "5.1.30"  
)
```

```
play.Project.playJavaSettings
```

## Appendix 7

File: \$root/server/app/models/database/Database.java

```
package models.database;

import models.datatypes.Airport;
import models.datatypes.Flight;

public interface Database {

    public long insertAirport(Airport airport);
    public long insertFlight(Flight flight);
    public long selectFlightByDepTime(int depTime);
    public long joinSelectFlightByDest(String dest);
}
```

## Appendix 8

File: \$root/server/app/models/datatypes/Airport.java

```
package models.datatypes;

import java.util.LinkedList;
import java.util.List;

public class Airport {
    private int id;
    private String iata;
    private String airport;
    private String city;
    private String state;
    private String country;
    private Double lat;
    private Double longitude;
    private List<Flight> flyingIn = new LinkedList<>();
    private List<Flight> flyingOut = new LinkedList<>();

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getIata() {
        return iata;
    }

    public void setIata(String iata) {
        this.iata = iata;
    }

    public String getAirport() {
        return airport;
    }

    public void setAirport(String airport) {
        this.airport = airport;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    public String getCountry() {
        return country;
    }
}
```

```
    public void setCountry(String country) {
        this.country = country;
    }

    public Double getLat() {
        return lat;
    }

    public void setLat(Double lat) {
        this.lat = lat;
    }

    public Double getLongitude() {
        return longitude;
    }

    public void setLongitude(Double longitude) {
        this.longitude = longitude;
    }

    public List<Flight> getFlyingIn() {
        return flyingIn;
    }

    public void setFlyingIn(List<Flight> flyingIn) {
        this.flyingIn = flyingIn;
    }

    public List<Flight> getFlyingOut() {
        return flyingOut;
    }

    public void setFlyingOut(List<Flight> flyingOut) {
        this.flyingOut = flyingOut;
    }
}
```

## Appendix 9

File: \$root/server/app/models/datatypes/Flight.java

```
package models.datatypes;

public class Flight {
    private int id;
    private int year;
    private int month;
    private int dayOfMonth;
    private int dayOfWeek;
    private int depTime;
    private int CRSDepTime;
    private int arrTime;
    private int CRSArrTime;
    private String uniqueCarrier;
    private int flightNum;
    private String tailNum;
    private int actualElapsedTime;
    private int CRSElapsedTime;
    private int airTime;
    private int arrDelay;
    private int depDelay;
    private Airport origin;
    private Airport dest;
    private int distance;
    private int taxiIn;
    private int taxiOut;
    private int cancelled;
    private String cancellationCode;
    private String diverted;
    private int carrierDelay;
    private int weatherDelay;
    private int NASDelay;
    private int securityDelay;
    private int lateAircraftDelay;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getMonth() {
        return month;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public int getDayOfMonth() {
        return dayOfMonth;
    }
}
```

```

public void setDayOfMonth(int dayOfMonth) {
    this.dayOfMonth = dayOfMonth;
}

public int getDayOfWeek() {
    return dayOfWeek;
}

public void setDayOfWeek(int dayOfWeek) {
    this.dayOfWeek = dayOfWeek;
}

public int getDepTime() {
    return depTime;
}

public void setDepTime(int depTime) {
    this.depTime = depTime;
}

public int getCRSDepTime() {
    return CRSDepTime;
}

public void setCRSDepTime(int CRSDepTime) {
    this.CRSDepTime = CRSDepTime;
}

public int getArrTime() {
    return arrTime;
}

public void setArrTime(int arrTime) {
    this.arrTime = arrTime;
}

public int getCRSArrTime() {
    return CRSArrTime;
}

public void setCRSArrTime(int CRSArrTime) {
    this.CRSArrTime = CRSArrTime;
}

public String getUniqueCarrier() {
    return uniqueCarrier;
}

public void setUniqueCarrier(String uniqueCarrier) {
    this.uniqueCarrier = uniqueCarrier;
}

public int getFlightNum() {
    return flightNum;
}

public void setFlightNum(int flightNum) {
    this.flightNum = flightNum;
}

public String getTailNum() {
    return tailNum;
}

public void setTailNum(String tailNum) {

```

```

        this.tailNum = tailNum;
    }

    public int getActualElapsedTime() {
        return actualElapsedTime;
    }

    public void setActualElapsedTime(int actualElapsedTime) {
        this.actualElapsedTime = actualElapsedTime;
    }

    public int getCRSElapsedTime() {
        return CRSElapsedTime;
    }

    public void setCRSElapsedTime(int CRSElapsedTime) {
        this.CRSElapsedTime = CRSElapsedTime;
    }

    public int getAirTime() {
        return airTime;
    }

    public void setAirTime(int airTime) {
        this.airTime = airTime;
    }

    public int getArrDelay() {
        return arrDelay;
    }

    public void setArrDelay(int arrDelay) {
        this.arrDelay = arrDelay;
    }

    public int getDepDelay() {
        return depDelay;
    }

    public void setDepDelay(int depDelay) {
        this.depDelay = depDelay;
    }

    public Airport getOrigin() {
        return origin;
    }

    public void setOrigin(Airport origin) {
        this.origin = origin;
    }

    public Airport getDest() {
        return dest;
    }

    public void setDest(Airport dest) {
        this.dest = dest;
    }

    public int getDistance() {
        return distance;
    }

    public void setDistance(int distance) {
        this.distance = distance;
    }
}

```

```

public int getTaxiIn() {
    return taxiIn;
}

public void setTaxiIn(int taxiIn) {
    this.taxiIn = taxiIn;
}

public int getTaxiOut() {
    return taxiOut;
}

public void setTaxiOut(int taxiOut) {
    this.taxiOut = taxiOut;
}

public int getCancelled() {
    return cancelled;
}

public void setCancelled(int cancelled) {
    this.cancelled = cancelled;
}

public String getCancellationCode() {
    return cancellationCode;
}

public void setCancellationCode(String cancellationCode) {
    this.cancellationCode = cancellationCode;
}

public String getDiverted() {
    return diverted;
}

public void setDiverted(String diverted) {
    this.diverted = diverted;
}

public int getCarrierDelay() {
    return carrierDelay;
}

public void setCarrierDelay(int carrierDelay) {
    this.carrierDelay = carrierDelay;
}

public int getWeatherDelay() {
    return weatherDelay;
}

public void setWeatherDelay(int weatherDelay) {
    this.weatherDelay = weatherDelay;
}

public int getNASDelay() {
    return NASDelay;
}

public void setNASDelay(int NASDelay) {
    this.NASDelay = NASDelay;
}

public int getSecurityDelay() {

```



```
        return securityDelay;
    }

    public void setSecurityDelay(int securityDelay) {
        this.securityDelay = securityDelay;
    }

    public int getLateAircraftDelay() {
        return lateAircraftDelay;
    }

    public void setLateAircraftDelay(int lateAircraftDelay) {
        this.lateAircraftDelay = lateAircraftDelay;
    }
}
```

## Appendix 10

```
package controllers;

import com.google.gson.Gson;
import models.database.DB4o;
import models.database.Database;
import models.database.Hibernate;
import models.database.MySQL;
import models.datatypes.Airport;
import models.datatypes.Flight;
import play.data.DynamicForm;
import play.data.Form;
import play.mvc.Controller;
import play.mvc.Result;

public class Insert extends Controller {

    public static Result oneFlight(Long time) {
        Long reqArrTime = System.currentTimeMillis();
        Long[] times = new Long[4];
        times[0] = time;
        times[1] = reqArrTime;

        DynamicForm requestData = Form.form().bindFromRequest();
        String json = requestData.get("data");
        Gson gson = new Gson();
        Flight flight = gson.fromJson(json, Flight.class);

        Database db = new MySQL();
        //Database db = new DB4o();
        //Database db = new Hibernate();

        times[2] = System.currentTimeMillis();
        times[3] = db.insertFlight(flight);

        if(times[3] == -1) return internalServerError();
        return ok "[" + times[0] + ", " + times[1] + ", " + times[2] + ", " + times[3]
+ "]"");
    }

    public static Result oneAirport(Long time) {
        Long reqArrTime = System.currentTimeMillis();
        Long[] times = new Long[4];
        times[0] = time;
        times[1] = reqArrTime;

        DynamicForm requestData = Form.form().bindFromRequest();
        String json = requestData.get("data");
        Gson gson = new Gson();
        Airport airport = gson.fromJson(json, Airport.class);

        Database db = new MySQL();
        //Database db = new DB4o();
        //Database db = new Hibernate();

        times[2] = System.currentTimeMillis();
        times[3] = db.insertAirport(airport);

        if (times[3] == -1) return internalServerError();
        return ok "[" + times[0] + ", " + times[1] + ", " + times[2] + ", " + times[3]
+ "]"");
    }
}
```

## Appendix 11

```
package controllers;

import models.database.DB4o;
import models.database.Database;
import models.database.Hibernate;
import models.database.MySQL;
import play.mvc.Controller;
import play.mvc.Result;

public class Select extends Controller {

    public static Result flightByDepTime(int depTime, Long time) {
        Long reqArrTime = System.currentTimeMillis();
        Long[] times = new Long[4];
        times[0] = time;
        times[1] = reqArrTime;

        //Database db = new MySQL();
        //Database db = new DB4o();
        Database db = new Hibernate();

        times[2] = System.currentTimeMillis();
        times[3] = db.selectFlightByDepTime(depTime);
        if (times[3] == -1) return internalServerError();
        return ok "[" + times[0] + ", " + times[1] + ", " + times[2] + ", " + times[3]
+ "]" );
    }

    public static Result joinFlightByDest(String dest, Long time) {
        Long reqArrTime = System.currentTimeMillis();
        Long[] times = new Long[4];
        times[0] = time;
        times[1] = reqArrTime;

        //Database db = new MySQL();
        //Database db = new DB4o();
        Database db = new Hibernate();

        times[2] = System.currentTimeMillis();
        times[3] = db.joinSelectFlightByDest(dest);
        if (times[3] == -1) return internalServerError();
        return ok "[" + times[0] + ", " + times[1] + ", " + times[2] + ", " + times[3]
+ "]" );
    }

}
```

## Appendix 12

File: \$root/server/app/models/database/MySQL.java

```
package models.database;

import models.datatypes.Airport;
import models.datatypes.Flight;
import play.db.DB;

import java.sql.*;

public class MySQL implements Database {
    Connection conn = null;

    public MySQL() {
        conn = DB.getConnection();
    }

    @Override
    public long insertAirport(Airport airport) {
        try {
            PreparedStatement pst = conn.prepareStatement("INSERT INTO airports
(iata, airport, city, state," +
                "country, lat, longitude) VALUES (?, ?, ?, ?, ?, ?, ?)");
            pst.setString(1, airport.getIata());
            pst.setString(2, airport.getAirport());
            pst.setString(3, airport.getCity());
            pst.setString(4, airport.getState());
            pst.setString(5, airport.getCountry());
            pst.setDouble(6, airport.getLat());
            pst.setDouble(7, airport.getLongitude());

            pst.executeUpdate();
            pst.close();
        } catch (SQLException e) {
            System.err.println("Could not create statement");
            e.printStackTrace();
            return -1;
        } finally {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        return System.currentTimeMillis();
    }

    @Override
    public long insertFlight(Flight flight) {
        try {
            PreparedStatement pst = conn.prepareStatement("INSERT INTO ontime (Year,
Month, DayOfMonth, DayOfWeek," +
                "DepTime, CRSDepTime, ArrTime, CRSArrTime, UniqueCarrier,
FlightNum, TailNum, ActualElapsedTime, " +
                "CRSElapsedTime, AirTime, ArrDelay, DepDelay, Origin,
Dest, Distance, TaxiIn, TaxiOut, Cancelled, " +
                "CancellationCode, Diverted, CarrierDelay, WeatherDelay,
NASDelay, SecurityDelay," +
```

```

        "LateAircraftDelay) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
" +
        "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
    pst.setInt(1, flight.getYear());
    pst.setInt(2, flight.getMonth());
    pst.setInt(3, flight.getDayOfMonth());
    pst.setInt(4, flight.getDayOfWeek());
    pst.setInt(5, flight.getDepTime());
    pst.setInt(6, flight.getCRSDepTime());
    pst.setInt(7, flight.getArrTime());
    pst.setInt(8, flight.getCRSArrTime());
    pst.setString(9, flight.getUniqueCarrier());
    pst.setInt(10, flight.getFlightNum());
    pst.setString(11, flight.getTailNum());
    pst.setInt(12, flight.getActualElapsedTime());
    pst.setInt(13, flight.getCRSElapsedTime());
    pst.setInt(14, flight.getAirTime());
    pst.setInt(15, flight.getArrDelay());
    pst.setInt(16, flight.getDepDelay());

    PreparedStatement originPst = conn.prepareStatement("SELECT id FROM
airports WHERE iata = ?");
    originPst.setString(1, flight.getOrigin().getIata());
    ResultSet originResult = originPst.executeQuery();
    originResult.next();
    pst.setInt(17, originResult.getInt("id"));
    originResult.close();
    originPst.close();

    PreparedStatement destPst = conn.prepareStatement("SELECT id FROM
airports WHERE iata = ?");
    destPst.setString(1, flight.getDest().getIata());
    ResultSet destResult = destPst.executeQuery();
    destResult.next();
    pst.setInt(18, destResult.getInt("id"));
    destResult.close();
    destPst.close();

    pst.setInt(19, flight.getDistance());
    pst.setInt(20, flight.getTaxiIn());
    pst.setInt(21, flight.getTaxiOut());
    pst.setInt(22, flight.getCancelled());
    pst.setString(23, flight.getCancellationCode());
    pst.setString(24, flight.getDiverted());
    pst.setInt(25, flight.getCarrierDelay());
    pst.setInt(26, flight.getWeatherDelay());
    pst.setInt(27, flight.getNASDelay());
    pst.setInt(28, flight.getSecurityDelay());
    pst.setInt(29, flight.getLateAircraftDelay());
    pst.executeUpdate();
    pst.close();
} catch (SQLException e) {
    System.err.println("Could not create statement");
    e.printStackTrace();
    return -1;
} finally {
    try {
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

return System.currentTimeMillis();
}

```

```

@Override
public long selectFlightByDepTime(int depTime) {
    try {
        PreparedStatement pst = conn.prepareStatement("SELECT * FROM ontime WHERE
DepTime = ?");
        pst.setInt(1, depTime);
        ResultSet result = pst.executeQuery();
        result.next();
        pst.close();
    } catch (SQLException e) {
        System.err.println("Could not create statement");
        e.printStackTrace();
        return -1;
    } finally {
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    return System.currentTimeMillis();
}

@Override
public long joinSelectFlightByDest(String dest) {
    try {
        PreparedStatement destPst = conn.prepareStatement("SELECT id FROM
airports WHERE iata = ?");
        destPst.setString(1, dest);
        ResultSet destResult = destPst.executeQuery();
        destResult.next();
        PreparedStatement pst = conn.prepareStatement("SELECT * FROM ontime WHERE
Dest = ?");
        pst.setInt(1, destResult.getInt("id"));
        ResultSet result = pst.executeQuery();
        result.next();
        pst.close();
    } catch (SQLException e) {
        System.err.println("Could not create statement");
        e.printStackTrace();
        return -1;
    } finally {
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    return System.currentTimeMillis();
}

public void purge() {
    try {
        Statement pst = conn.createStatement();
        pst.execute("DELETE FROM ontime WHERE 1=1"); // Truncate is not supported
when using foreign keys
        pst.execute("DELETE FROM airports WHERE 1=1");
        pst.close();
    } catch (SQLException e) {
        System.err.println("Could not create statement");
        e.printStackTrace();
    } finally {
        try {
            conn.close();

```

```
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

## Appendix 13

File: \$root/server/app/models/database/Hibernate.java

```
package models.database;

import models.datatypes.Airport;
import models.datatypes.Flight;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import play.Logger;

import java.util.List;

public class Hibernate implements Database {
    private static SessionFactory sessionFactory;
    private static ServiceRegistry serviceRegistry;

    public static void createSessionFactory() {
        Configuration configuration = new Configuration();
        configuration.configure();
        serviceRegistry = new StandardServiceRegistryBuilder().applySettings(
            configuration.getProperties()).build();
        sessionFactory = configuration.buildSessionFactory(serviceRegistry);
        Logger.info("Hibernate sessionFactory created");
    }

    public static void closeSessionFactory() {
        sessionFactory.close();
        Logger.info("Hibernate sessionFactory closed");
    }

    @Override
    public long insertAirport(Airport airport) {
        Session session = sessionFactory.openSession();
        session.beginTransaction();
        session.save(airport);
        session.getTransaction().commit();
        session.close();
        return System.currentTimeMillis();
    }

    @Override
    public long insertFlight(Flight flight) {
        Session session = sessionFactory.openSession();

        Query query = session.createQuery("from Airport where iata = :iata ");
        query.setParameter("iata", flight.getDest().getIata());
        List list = query.list();

        Airport destAirport = (Airport) list.get(0);
        flight.setDest(destAirport);

        query = session.createQuery("from Airport where iata = :iata ");
        query.setParameter("iata", flight.getOrigin().getIata());
        list = query.list();

        Airport origAirport = (Airport) list.get(0);
        flight.setOrigin(origAirport);

        session.beginTransaction();
    }
}
```



```

        session.save(flight);

        session.getTransaction().commit();
        session.close();
        return System.currentTimeMillis();
    }

    @Override
    public long selectFlightByDepTime(int depTime) {
        Session session = sessionFactory.openSession();
        Query query = session.createQuery("from Flight where depTime = :depTime ");
        query.setParameter("depTime", depTime);
        List list = query.list();
        int size = list.size();
        session.close();
        return System.currentTimeMillis();
    }

    @Override
    public long joinSelectFlightByDest(String dest) {
        Session session = sessionFactory.openSession();
        Query query = session.createQuery("from Airport where iata = :iata ");
        query.setParameter("iata", dest);
        List list = query.list();
        Airport airport = (Airport) list.get(0);
        int size = airport.getFlyingIn().size();
        session.close();
        return System.currentTimeMillis();
    }
}

```

## Appendix 14

File: \$root/server/app/Global.java

```
import models.database.DB4o;
import models.database.Hibernate;
import play.*;

public class Global extends GlobalSettings {

    @Override
    public void onStart(Application app) {
        Logger.info("Application has started");
        DB4o.openDB();
        Hibernate.createSessionFactory();
    }

    @Override
    public void onStop(Application app) {
        Logger.info("Application shutdown...");
        DB4o.closeDB();
        Hibernate.closeSessionFactory();
    }

}
```

## Appendix 15

File: \$root/server/conf/hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>

    <property name="hibernate.connection.datasource">DefaultDS</property>

    <mapping resource="Airport.hbm.xml"/>
    <mapping resource="Flight.hbm.xml"/>

  </session-factory>
</hibernate-configuration>
```

## Appendix 16

```
File: $root/server/conf/Flight.hbm.xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="models.datatypes.Flight" table="ontime">
        <meta attribute="class-description">
            This class contains the flight.
        </meta>
        <id name="id" type="int" column="Id">
            <generator class="increment"/>
        </id>
        <property name="year" column="Year"/>
        <property name="month" column="Month"/>
        <property name="dayOfMonth" column="DayOfMonth"/>
        <property name="dayOfWeek" column="DayOfWeek"/>
        <property name="depTime" column="DepTime"/>
        <property name="CRSDepTime" column="CRSDepTime"/>
        <property name="arrTime" column="ArrTime"/>
        <property name="CRSArrTime" column="CRSArrTime"/>
        <property name="uniqueCarrier" column="UniqueCarrier"/>
        <property name="flightNum" column="FlightNum"/>
        <property name="tailNum" column="TailNum"/>
        <property name="actualElapsedTime" column="ActualElapsedTime"/>
        <property name="CRSElapsedTime" column="CRSElapsedTime"/>
        <property name="airTime" column="AirTime"/>
        <property name="arrDelay" column="ArrDelay"/>
        <property name="depDelay" column="DepDelay"/>
        <many-to-one name="origin" column="Origin"
            class="models.datatypes.Airport"/>
        <many-to-one name="dest" column="Dest"
            class="models.datatypes.Airport"/>
        <property name="distance" column="Distance"/>
        <property name="taxiIn" column="TaxiIn"/>
        <property name="taxiOut" column="TaxiOut"/>
        <property name="cancelled" column="Cancelled"/>
        <property name="cancellationCode" column="CancellationCode"/>
        <property name="diverted" column="Diverted"/>
        <property name="carrierDelay" column="CarrierDelay"/>
        <property name="weatherDelay" column="WeatherDelay"/>
        <property name="NASDelay" column="NASDelay"/>
        <property name="securityDelay" column="SecurityDelay"/>
        <property name="lateAircraftDelay" column="LateAircraftDelay"/>
    </class>
</hibernate-mapping>
```

## Appendix 17

```
File: $root/server/conf/Airport.hbm.xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="models.datatypes.Airport" table="airports">
        <meta attribute="class-description">
            This class contains the airport detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="increment"/>
        </id>
        <property name="iata" column="iata"/>
        <property name="airport" column="airport"/>
        <property name="city" column="city"/>
        <property name="state" column="state"/>
        <property name="country" column="country"/>
        <property name="lat" column="lat"/>
        <property name="longitude" column="longitude"/>
        <bag name="flyingIn" cascade="all" inverse="true">
            <key column="Dest"/>
            <one-to-many class="models.datatypes.Flight"/>
        </bag>
        <bag name="flyingOut" cascade="all" inverse="true">
            <key column="Origin"/>
            <one-to-many class="models.datatypes.Flight"/>
        </bag>
    </class>
</hibernate-mapping>
```

## Appendix 18

File: \$root/server/app/models/database/DB4o.java

```
package models.database;

import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.reflect.generic.GenericClass;
import com.db4o.reflect.generic.GenericObject;
import com.db4o.reflect.generic.GenericReflector;
import models.datatypes.Airport;
import models.datatypes.Flight;
import play.Logger;

import java.util.HashSet;
import java.util.LinkedList;

public class DB4o implements Database {
    private static ObjectContainer db;

    public static void openDB() {
        db = Db4oEmbedded.openFile(Db4oEmbedded
            .newConfiguration(), "db4o.db");
        Logger.info("DB4o opened");
    }

    public static void closeDB() {
        db.close();
        Logger.info("DB4o closed");
    }

    @Override
    public long insertAirport(Airport airport) {
        try {
            db.store(airport);
            db.commit();
        } catch (Exception ex) {
            ex.printStackTrace();
            return -1;
        }
        return System.currentTimeMillis();
    }

    @Override
    public long insertFlight(Flight flight) {
        try {
            ObjectSet result = db.queryByExample(flight.getDest());
            Airport destAirport = (Airport) result.next();

            destAirport.getFlyingIn().add(flight);
            flight.setDest(destAirport);

            result = db.queryByExample(flight.getOrigin());
            Airport originAirport = (Airport) result.next();

            originAirport.getFlyingOut().add(flight);

            flight.setOrigin(originAirport);

            db.store(flight);
            db.store(destAirport);
            db.store(originAirport);
            db.commit();
        }
    }
}
```

```

        } catch (Exception ex) {
            ex.printStackTrace();
            return -1;
        }
        return System.currentTimeMillis();
    }

    @Override
    public long selectFlightByDepTime(int depTime) {

        try {
            Flight flightExample = new Flight();
            flightExample.setDepTime(depTime);
            ObjectSet result = db.queryByExample(flightExample);
            int size = result.size();
        } catch (Exception ex) {
            ex.printStackTrace();
            return -1;
        }
        return System.currentTimeMillis();
    }

    @Override
    public long joinSelectFlightByDest(String dest) {
        Airport destAirport = new Airport();
        destAirport.setIata(dest);
        ObjectSet result = db.queryByExample(destAirport);
        destAirport = (Airport) result.next();
        int size = destAirport.getFlyingIn().size();
        return System.currentTimeMillis();
    }

    public void purge() {
        closeDB();
        File dbfile = new File("db4o.db");
        dbfile.delete();
        openDB();
    }
}

```

## Appendix 19

File: \$root/client/src/net/pixomania/dbtest/client/models/Util.java

```
package net.pixomania.dbtest.client.models;

import net.pixomania.dbtest.client.datatypes.Airport;
import net.pixomania.dbtest.client.datatypes.Flight;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.LinkedList;
import java.util.Scanner;

public class Util {

    public static LinkedList<Flight> loadFlights(File csv) throws FileNotFoundException {
        LinkedList<Flight> flights = new LinkedList<>();
        Scanner scanner = new Scanner(csv);
        scanner.useDelimiter(",");
        scanner.nextLine();
        while(scanner.hasNextLine()) {
            Flight flight = new Flight();

            Scanner scanner1 = new Scanner(scanner.nextLine());
            scanner1.useDelimiter(",");
            flight.setYear(checkInt(scanner1.next()));
            flight.setMonth(checkInt(scanner1.next()));
            flight.setDayOfMonth(checkInt(scanner1.next()));
            flight.setDayOfWeek(checkInt(scanner1.next()));
            flight.setDepTime(checkInt(scanner1.next()));
            flight.setCRSDepTime(checkInt(scanner1.next()));
            flight.setArrTime(checkInt(scanner1.next()));
            flight.setCRSArrTime(checkInt(scanner1.next()));
            flight.setUniqueCarrier(checkString(scanner1.next()));
            flight.setFlightNum(checkInt(scanner1.next()));
            flight.setTailNum(checkString(scanner1.next()));
            flight.setActualElapsedTime(checkInt(scanner1.next()));
            flight.setCRSElapsedTime(checkInt(scanner1.next()));
            flight.setAirTime(checkInt(scanner1.next()));
            flight.setArrDelay(checkInt(scanner1.next()));
            flight.setDepDelay(checkInt(scanner1.next()));
            Airport origin = new Airport();
            origin.setIata(checkString(scanner1.next()));
            flight.setOrigin(origin);
            Airport dest = new Airport();
            dest.setIata(checkString(scanner1.next()));
            flight.setDest(dest);
            flight.setDistance(checkInt(scanner1.next()));
            flight.setTaxiIn(checkInt(scanner1.next()));
            flight.setTaxiOut(checkInt(scanner1.next()));
            flight.setCancelled(checkInt(scanner1.next()));
            flight.setCancellationCode(checkString(scanner1.next()));
            flight.setDiverted(checkString(scanner1.next()));
            flight.setCarrierDelay(checkInt(scanner1.next()));
            flight.setWeatherDelay(checkInt(scanner1.next()));
            flight.setNASDelay(checkInt(scanner1.next()));
            flight.setSecurityDelay(checkInt(scanner1.next()));
            flight.setLateAircraftDelay(checkInt(scanner1.next()));

            flights.add(flight);
            if(flights.size() == 3000) break;
        }

        return flights;
    }
}
```



```

    }

    private static int checkInt(String input){
        try {
            return Integer.parseInt(input);
        } catch (NumberFormatException ex){
            return -1;
        }
    }

    private static String checkString(String input) {
        if(input.equals("NA")) return "";
        return input;
    }

    public static LinkedList<Airport> loadAirports(File csv) throws FileNotFoundException
    {
        LinkedList<Airport> airports = new LinkedList<>();
        Scanner scanner = new Scanner(csv);
        scanner.useDelimiter(",");
        scanner.nextLine();
        while (scanner.hasNextLine()) {
            Airport airport = new Airport();
            String line = scanner.nextLine();
            Scanner scanner1 = new Scanner(line);
            scanner1.useDelimiter(",");
            airport.setIata(scanner1.next());
            airport.setAirport(scanner1.next());
            airport.setCity(scanner1.next());
            airport.setState(scanner1.next());
            airport.setCountry(scanner1.next());
            airport.setLat(Double.parseDouble(scanner1.next()));
            airport.setLongitude(Double.parseDouble(scanner1.next()));

            airports.add(airport);
        }

        return airports;
    }
}

```

## Appendix 20

File: \$root/client/src/net/pixomania/dbtest/client/http/HttpClient.java

```
package net.pixomania.dbtest.client.http;

import com.google.gson.Gson;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Arrays;
import java.util.Collections;

public class HttpClient {

    public static long totaltime = 0;
    public static long requestsmade = 0;
    public static long high = 0;
    public static long low = 50000;

    public static void sendPOST(String url, String params) throws IOException {
        java.net.URL obj = new URL(url);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();

        //add request header
        con.setRequestMethod("POST");
        con.setRequestProperty("User-Agent", "Mozilla/5.0");
        con.setRequestProperty("Accept-Language", "en-US,en;q=0.5");

        // Send post request
        con.setDoOutput(true);
        DataOutputStream wr = new DataOutputStream(con.getOutputStream());
        wr.writeBytes("data=" + params);
        wr.flush();
        wr.close();

        int responseCode = con.getResponseCode();

        BufferedReader in = new BufferedReader(
            new InputStreamReader(con.getInputStream()));
        String inputLine;
        StringBuffer response = new StringBuffer();

        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        Long timeTaken = System.currentTimeMillis();
        in.close();

        Gson gson = new Gson();

        Long[] longArr = gson.fromJson(response.toString(), Long[].class);
        Long[] newArr = Arrays.copyOf(longArr, longArr.length+1);
        newArr[newArr.length-1] = timeTaken;

        long timetaken = (newArr[newArr.length - 1] - newArr[0]);
        System.out.println(gson.toJson(newArr) + " Request time: " + timetaken + "ms");
        totaltime += timetaken;
        requestsmade++;

        if(timetaken < low) {
```

```

        low = timetaken;
    } else if(timetaken > high) {
        high = timetaken;
    }
}

public static void sendGET(String url) throws IOException {
    java.net.URL obj = new URL(url);
    HttpURLConnection con = (HttpURLConnection) obj.openConnection();

    //add request header
    con.setRequestMethod("GET");
    con.setRequestProperty("User-Agent", "Mozilla/5.0");
    con.setRequestProperty("Accept-Language", "en-US,en;q=0.5");

    int responseCode = con.getResponseCode();

    BufferedReader in = new BufferedReader(
        new InputStreamReader(con.getInputStream()));
    String inputLine;
    StringBuffer response = new StringBuffer();

    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    Long timeTaken = System.currentTimeMillis();
    in.close();

    Gson gson = new Gson();

    Long[] longArr = gson.fromJson(response.toString(), Long[].class);
    Long[] newArr = Arrays.copyOf(longArr, longArr.length + 1);
    newArr[newArr.length - 1] = timeTaken;

    long timetaken = (newArr[newArr.length - 1] - newArr[0]);
    System.out.println(gson.toJson(newArr) + " Request time: " + timetaken + "ms");
    totaltime += timetaken;
    requestsmade++;

    if (timetaken < low) {
        low = timetaken;
    } else if (timetaken > high) {
        high = timetaken;
    }
}
}

```

## Appendix 21

File: \$root/client/src/net/pixomania/dbtest/client/Main.java

```
package net.pixomania.dbtest.client;

import com.google.gson.Gson;
import net.pixomania.dbtest.client.datatypes.Airport;
import net.pixomania.dbtest.client.datatypes.Flight;
import net.pixomania.dbtest.client.http.HttpClient;
import net.pixomania.dbtest.client.models.Util;

import java.io.File;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        try {

            LinkedList<Airport> airports = Util.loadAirports(new
File("C:\\Users\\Victor\\dbtest\\client\\data\\airports.csv"));
            Gson gson = new Gson();
            DecimalFormat df = new DecimalFormat("#.000");
            for (Airport airport : airports) {
                String json = gson.toJson(airport);
                String url = "http://localhost:9000/insert/airport/one/" +
System.currentTimeMillis();
                HttpClient.sendPOST(url, json);
                System.out.println(HttpClient.totaltime + "ms | " +
df.format((double)HttpClient.totaltime/(double)1000) + "s " +
"| average (ms): " + df.format((double)
HttpClient.totaltime / (double) HttpClient.requestsmade) + " | requests made: " +
HttpClient.requestsmade +
" | low: " + HttpClient.low + "ms | high: " +
HttpClient.high + "ms");
            }

            /*
            LinkedList<Flight> flights = Util.loadFlights(new
File("C:\\Users\\Victor\\dbtest\\client\\data\\1987.csv"));
            Gson gson = new Gson();
            DecimalFormat df = new DecimalFormat("#.000");
            for (Flight flight : flights) {
                String json = gson.toJson(flight);
                String url = "http://localhost:9000/insert/flight/one/" +
System.currentTimeMillis();
                HttpClient.sendPOST(url, json);
                System.out.println(HttpClient.totaltime + "ms | " +
df.format((double) HttpClient.totaltime / (double) 1000) + "s " +
"| average (ms): " + df.format((double)
HttpClient.totaltime / (double) HttpClient.requestsmade) + " | requests made: " +
HttpClient.requestsmade +
" | low: " + HttpClient.low + "ms | high: " +
HttpClient.high + "ms");
            }
            */
            /*
            LinkedList<Flight> flights = Util.loadFlights(new
File("C:\\Users\\Victor\\dbtest\\client\\data\\1987.csv"));
```

```

        Gson gson = new Gson();
        DecimalFormat df = new DecimalFormat("#.000");
        for (Flight flight : flights) {

            String url = "http://localhost:9000/select/flight/deptime/" +
flight.getDepTime() + "/" + System.currentTimeMillis();
            HttpClient.sendGET(url);
            System.out.println(HttpClient.totaltime + "ms | " +
df.format((double) HttpClient.totaltime / (double) 1000) + "s " +
            "| average (ms): " + df.format((double)
HttpClient.totaltime / (double) HttpClient.requestsmade) + " | requests made: " +
HttpClient.requestsmade +
            " | low: " + HttpClient.low + "ms | high: " +
HttpClient.high + "ms");
        }
    /*
    /*
        LinkedList<Flight> flights = Util.loadFlights(new
File("C:\\Users\\Victor\\dbtest\\client\\data\\1987.csv"));
        Gson gson = new Gson();
        DecimalFormat df = new DecimalFormat("#.000");
        for (Flight flight : flights) {

            String url = "http://localhost:9000/select/join/flight/dest/" +
flight.getDest().getIata() + "/" + System.currentTimeMillis();
            HttpClient.sendGET(url);
            System.out.println(HttpClient.totaltime + "ms | " +
df.format((double) HttpClient.totaltime / (double) 1000) + "s " +
            "| average (ms): " + df.format((double)
HttpClient.totaltime / (double) HttpClient.requestsmade) + " | requests made: " +
HttpClient.requestsmade +
            " | low: " + HttpClient.low + "ms | high: " +
HttpClient.high + "ms");
        }
    /*
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

## Appendix 22

```
-- phpMyAdmin SQL Dump
-- version 4.0.4
-- http://www.phpmyadmin.net
--
-- Vård: localhost
-- Skapad: 19 apr 2014 kl 19:32
-- Serverversion: 5.6.12-log
-- PHP-version: 5.4.3

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;

--
-- Databas: `testdb`
--
CREATE DATABASE IF NOT EXISTS `testdb` DEFAULT CHARACTER SET latin1 COLLATE latin1_swedish_ci;
USE `testdb`;

--
-- Tabellstruktur `airports`
--

CREATE TABLE IF NOT EXISTS `airports` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `iata` varchar(10) NOT NULL,
  `airport` varchar(64) NOT NULL,
  `city` varchar(64) NOT NULL,
  `state` varchar(20) NOT NULL,
  `country` varchar(42) NOT NULL,
  `lat` double NOT NULL,
  `longitude` double NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 ;

--
-- Tabellstruktur `ontime`
--

CREATE TABLE IF NOT EXISTS `ontime` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `Year` int(11) DEFAULT NULL,
  `Month` int(11) DEFAULT NULL,
  `DayOfMonth` int(11) DEFAULT NULL,
  `DayOfWeek` int(11) DEFAULT NULL,
  `DepTime` int(11) DEFAULT NULL,
  `CRSDepTime` int(11) DEFAULT NULL,
  `ArrTime` int(11) DEFAULT NULL,
  `CRSArrTime` int(11) DEFAULT NULL,
  `UniqueCarrier` varchar(5) DEFAULT NULL,
  `FlightNum` int(11) DEFAULT NULL,
  `TailNum` varchar(8) DEFAULT NULL,
  `ActualElapsedTime` int(11) DEFAULT NULL,
  `CRSElapsedTime` int(11) DEFAULT NULL,
```

```

`AirTime` int(11) DEFAULT NULL,
`ArrDelay` int(11) DEFAULT NULL,
`DepDelay` int(11) DEFAULT NULL,
`Origin` int(11) DEFAULT NULL,
`Dest` int(11) DEFAULT NULL,
`Distance` int(11) DEFAULT NULL,
`TaxiIn` int(11) DEFAULT NULL,
`TaxiOut` int(11) DEFAULT NULL,
`Cancelled` int(11) DEFAULT NULL,
`CancellationCode` varchar(1) DEFAULT NULL,
`Diverted` varchar(1) DEFAULT NULL,
`CarrierDelay` int(11) DEFAULT NULL,
`WeatherDelay` int(11) DEFAULT NULL,
`NASDelay` int(11) DEFAULT NULL,
`SecurityDelay` int(11) DEFAULT NULL,
`LateAircraftDelay` int(11) DEFAULT NULL,
PRIMARY KEY (`Id`),
KEY `Origin` (`Origin`),
KEY `Dest` (`Dest`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 ;

--
-- Restriktioner för dumpade tabeller
--

--
-- Restriktioner för tabell `ontime`
--
ALTER TABLE `ontime`
  ADD CONSTRAINT `ontime_ibfk_2` FOREIGN KEY (`Dest`) REFERENCES `airports` (`id`) ON DELETE
  CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `ontime_ibfk_1` FOREIGN KEY (`Origin`) REFERENCES `airports` (`id`) ON DELETE
  CASCADE ON UPDATE CASCADE;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;

```

## Appendix 23

File: \$root/client/src/net/pixomania/dbtest/client/models/ConnectionPool.java

```
package net.pixomania.dbtest.client.models;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class ConnectionPool {
    private static List<Connection> pool = new LinkedList<Connection>();

    public static void open(int connections){
        for(int i = 0; i < connections; i++) {
            try {

pool.add(DriverManager.getConnection("jdbc:mysql://localhost:3306/result",      "root",
""));
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Opened " + connections + " connections");
        }

        public synchronized static Connection getConnection() {
            return pool.remove(0);
        }

        public synchronized static void returnConnection(Connection con) {
            pool.add(con);
        }

        public static void close() {
            for(Connection con : pool) {
                try {
                    con.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```



## Appendix 24

File: \$root/client/src/net/pixomania/dbtest/client/http/HttpClient.java

```
package net.pixomania.dbtest.client.http;

import com.google.gson.Gson;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Arrays;
import java.util.Collections;

public class HttpClient {

    public static long[] sendPOST(String url, String params) throws IOException {
        java.net.URL obj = new URL(url);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();

        //add request header
        con.setRequestMethod("POST");
        con.setRequestProperty("User-Agent", "Mozilla/5.0");
        con.setRequestProperty("Accept-Language", "en-US,en;q=0.5");

        // Send post request
        con.setDoOutput(true);
        DataOutputStream wr = new DataOutputStream(con.getOutputStream());
        wr.writeBytes("data=" + params);
        wr.flush();
        wr.close();

        int responseCode = con.getResponseCode();

        BufferedReader in = new BufferedReader(
            new InputStreamReader(con.getInputStream()));
        String inputLine;
        StringBuffer response = new StringBuffer();

        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        long timeTaken = System.currentTimeMillis();
        in.close();

        Gson gson = new Gson();

        long[] longArr = gson.fromJson(response.toString(), long[].class);
        long[] newArr = Arrays.copyOf(longArr, longArr.length+1);
        newArr[newArr.length-1] = timeTaken;

        return newArr;
    }

    public static long[] sendGET(String url) throws IOException {
        java.net.URL obj = new URL(url);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();

        //add request header
        con.setRequestMethod("GET");
        con.setRequestProperty("User-Agent", "Mozilla/5.0");
        con.setRequestProperty("Accept-Language", "en-US,en;q=0.5");
```

```

        int responseCode = con.getResponseCode();

        BufferedReader in = new BufferedReader(
            new InputStreamReader(con.getInputStream()));
        String inputLine;
        StringBuffer response = new StringBuffer();

        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        long timeTaken = System.currentTimeMillis();
        in.close();

        Gson gson = new Gson();

        long[] longArr = gson.fromJson(response.toString(), long[].class);
        long[] newArr = Arrays.copyOf(longArr, longArr.length + 1);
        newArr[newArr.length - 1] = timeTaken;

        return newArr;
    }
}

```

## Appendix 25

File: \$root/client/src/net/pixomania/dbtest/client/models/Test.java

```
package net.pixomania.dbtest.client.models;

import net.pixomania.dbtest.client.Main;

import java.sql.*;

public class Test {
    protected int testid;

    public Test(int type) {
        Connection conn = ConnectionPool.getConnection();
        PreparedStatement pst = null;
        try {
            pst = conn.prepareStatement("INSERT INTO tests (date, type, database_type)
VALUES (?, ?, ?)", Statement.RETURN_GENERATED_KEYS);
            pst.setTimestamp(1, new
java.sql.Timestamp(System.currentTimeMillis()));
            pst.setInt(2, type);
            pst.setInt(3, Main.testing_database);
            pst.executeUpdate();
            ResultSet rs = pst.getGeneratedKeys();
            if (rs.next()) testid = rs.getInt(1);
            pst.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        ConnectionPool.returnConnection(conn);
    }
}
```

## Appendix 26

File: \$root/client/src/net/pixomania/dbtest/client/models/InsertFlightTest.java

```
package net.pixomania.dbtest.client.models;

import com.google.gson.Gson;
import net.pixomania.dbtest.client.Main;
import net.pixomania.dbtest.client.datatypes.Flight;
import net.pixomania.dbtest.client.http.HttpClient;

import java.io.IOException;
import java.sql.*;
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class InsertFlightTest extends Test {

    public InsertFlightTest() {
        super(2);
    }

    public void run(LinkedList<Flight> flights) {
        ExecutorService pool = Executors.newFixedThreadPool(100);
        Gson gson = new Gson();
        for (Flight flight : flights) {
            String json = gson.toJson(flight);
            String url = Main.url + "/insert/flight/one/" + System.currentTimeMillis()
+ "/" + Main.testing_database;
            try {
                final long[] response = HttpClient.sendPOST(url, json);
                pool.submit(new Runnable() {
                    @Override
                    public void run() {
                        Connection conn = ConnectionPool.getConnection();
                        PreparedStatement pst = null;
                        try {
                            pst = conn.prepareStatement("INSERT INTO
responses (test_id) VALUES (?)", Statement.RETURN_GENERATED_KEYS);
                            pst.setInt(1, testid);
                            pst.executeUpdate();
                            ResultSet rs = pst.getGeneratedKeys();
                            int responseid = -1;
                            if (rs.next()) responseid = rs.getInt(1);
                            pst.close();

                            for (int i = 0; i < response.length; i++)
                                {
                                    pst
                                    =
conn.prepareStatement("INSERT INTO responses_parts (response_id, array_index, timevalue)
VALUES (?, ?, ?)");
                                    pst.setInt(1, responseid);
                                    pst.setInt(2, i);
                                    pst.setLong(3, response[i]);
                                    pst.executeUpdate();
                                    pst.close();
                                }
                        } catch (SQLException e) {
                            e.printStackTrace();
                        } finally {
                            ConnectionPool.returnConnection(conn);
                        }
                    }
                });
            }
        }
    }
}
```

```
        }  
        });  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
try {  
    pool.shutdown();  
    pool.awaitTermination(1, TimeUnit.DAYS);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
}
```

## Appendix 27

File: \$root/client/src/net/pixomania/dbtest/client/models/InsertAirportTest.java

```
package net.pixomania.dbtest.client.models;

import com.google.gson.Gson;
import net.pixomania.dbtest.client.Main;
import net.pixomania.dbtest.client.datatypes.Airport;
import net.pixomania.dbtest.client.http.HttpClient;
import java.io.IOException;
import java.sql.*;
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class InsertAirportTest extends Test {

    public InsertAirportTest() {
        super(1);
    }

    public void run(LinkedList<Airport> airports) {
        ExecutorService pool = Executors.newFixedThreadPool(100);
        Gson gson = new Gson();
        for (Airport airport : airports) {
            String json = gson.toJson(airport);
            String url = Main.url + "/insert/airport/one/" +
                System.currentTimeMillis() + "/" + Main.testing_database;
            try {
                final long[] response = HttpClient.sendPOST(url, json);
                pool.submit(new Runnable() {
                    @Override
                    public void run() {
                        Connection conn = ConnectionPool.getConnection();
                        PreparedStatement pst = null;
                        try {
                            pst = conn.prepareStatement("INSERT INTO
responses (test_id) VALUES (?)", Statement.RETURN_GENERATED_KEYS);
                            pst.setInt(1, testid);
                            pst.executeUpdate();
                            ResultSet rs = pst.getGeneratedKeys();
                            int responseid = -1;
                            if(rs.next()) responseid = rs.getInt(1) ;
                            pst.close();

                            for(int i = 0; i < response.length; i++) {
                                pst
                                =
conn.prepareStatement("INSERT INTO responses_parts (response_id, array_index, timevalue)
VALUES (?, ?, ?)");

                                pst.setInt(1, responseid);
                                pst.setInt(2, i);
                                pst.setLong(3, response[i]);
                                pst.executeUpdate();
                                pst.close();
                            }
                        } catch (SQLException e) {
                            e.printStackTrace();
                        }
                        ConnectionPool.returnConnection(conn);
                    }
                });
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
        }  
    }  
    try {  
        pool.shutdown();  
        pool.awaitTermination(1, TimeUnit.DAYS);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
}
```

## Appendix 28

File: \$root/client/src/net/pixomania/dbtest/client/models/ListFlightTest.java

```
package net.pixomania.dbtest.client.models;

import com.google.gson.Gson;
import net.pixomania.dbtest.client.Main;
import net.pixomania.dbtest.client.datatypes.Airport;
import net.pixomania.dbtest.client.http.HttpClient;

import java.io.IOException;
import java.sql.*;
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ListFlightTest extends Test {

    public ListFlightTest() {
        super(4);
    }

    public void run(LinkedList<Airport> airports) {
        ExecutorService pool = Executors.newFixedThreadPool(100);
        Gson gson = new Gson();
        for (Airport airport : airports) {
            String url = Main.url + "/select/join/flight/dest/" + airport.getIata()
+ "/" + System.currentTimeMillis() + "/" + Main.testing_database;
            try {
                final long[] response = HttpClient.sendGET(url);
                pool.submit(new Runnable() {
                    @Override
                    public void run() {
                        Connection conn = ConnectionPool.getConnection();
                        PreparedStatement pst = null;
                        try {
                            pst = conn.prepareStatement("INSERT INTO
responses (test_id) VALUES (?)", Statement.RETURN_GENERATED_KEYS);
                            pst.setInt(1, testid);
                            pst.executeUpdate();
                            ResultSet rs = pst.getGeneratedKeys();
                            int responseid = -1;
                            if (rs.next()) responseid = rs.getInt(1);
                            pst.close();

                            for (int i = 0; i < response.length; i++)
                                {
                                    pst
                                    =
conn.prepareStatement("INSERT INTO responses_parts (response_id, array_index, timevalue)
VALUES (?, ?, ?)");
                                    pst.setInt(1, responseid);
                                    pst.setInt(2, i);
                                    pst.setLong(3, response[i]);
                                    pst.executeUpdate();
                                    pst.close();
                                }
                        } catch (SQLException e) {
                            e.printStackTrace();
                        }
                        ConnectionPool.returnConnection(conn);
                    }
                });
            } catch (IOException e) {
```



```
        e.printStackTrace();
    }
    try {
        pool.shutdown();
        pool.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

## Appendix 29

File: \$root/client/src/net/pixomania/dbtest/client/models/SelectFlightTest.java

```
package net.pixomania.dbtest.client.models;

import com.google.gson.Gson;
import net.pixomania.dbtest.client.Main;
import net.pixomania.dbtest.client.datatypes.Flight;
import net.pixomania.dbtest.client.http.HttpClient;

import java.io.IOException;
import java.sql.*;
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class SelectFlightTest extends Test {

    public SelectFlightTest() {
        super(3);
    }

    public void run(LinkedList<Flight> flights) {
        ExecutorService pool = Executors.newFixedThreadPool(100);
        Gson gson = new Gson();
        for (Flight flight : flights) {
            String url = Main.url + "/select/flight/deptime/" + flight.getDepTime()
+ "/" + System.currentTimeMillis() + "/" + Main.testing_database;
            try {
                final long[] response = HttpClient.sendGET(url);
                pool.submit(new Runnable() {
                    @Override
                    public void run() {
                        Connection conn = ConnectionPool.getConnection();
                        PreparedStatement pst = null;
                        try {
                            pst = conn.prepareStatement("INSERT INTO
responses (test_id) VALUES (?)", Statement.RETURN_GENERATED_KEYS);
                            pst.setInt(1, testid);
                            pst.executeUpdate();
                            ResultSet rs = pst.getGeneratedKeys();
                            int responseid = -1;
                            if (rs.next()) responseid = rs.getInt(1);
                            pst.close();

                            for (int i = 0; i < response.length; i++)
                            {
                                pst
                                =
conn.prepareStatement("INSERT INTO responses_parts (response_id, array_index, timevalue)
VALUES (?, ?, ?)");
                                pst.setInt(1, responseid);
                                pst.setInt(2, i);
                                pst.setLong(3, response[i]);
                                pst.executeUpdate();
                                pst.close();
                            }
                        } catch (SQLException e) {
                            e.printStackTrace();
                        } finally {
                            ConnectionPool.returnConnection(conn);
                        }
                    }
                });
            }
        }
    }
}
```

```
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    try {  
        pool.shutdown();  
        pool.awaitTermination(1, TimeUnit.DAYS);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

## Appendix 30

File: \$root/client/src/net/pixomania/dbtest/client/Main.java

```
package net.pixomania.dbtest.client;

import net.pixomania.dbtest.client.datatypes.Airport;
import net.pixomania.dbtest.client.datatypes.Flight;
import net.pixomania.dbtest.client.http.HttpClient;
import net.pixomania.dbtest.client.models.*;
import java.io.File;
import java.util.Date;
import java.util.LinkedList;

public class Main {
    public static int testing_database = 1; // 1 = mysql, 2 = db4o, 3 = hibernate
    public static String url = "http://192.168.1.3:9000";

    public static void main(String[] args) {
        try {
            LinkedList<Airport> airports = Util.loadAirports(new
File("airports.csv"), 3376);
            ConnectionPool.open(100);
            for(int dataamount = 0; dataamount < 3; dataamount++) {
                for (int repetitions = 0; repetitions < 4; repetitions++) {
                    for (int database = 1; database < 4; database++) {
                        testing_database = database;
                        int toload = 0;

                        switch (dataamount) {
                            case 0:
                                toload = 1000;
                                break;
                            case 1:
                                toload = 10000;
                                break;
                            case 2:
                                toload = 20000;
                                break;
                            case 3:
                                toload = 80000;
                                break;
                        }

                        InsertAirportTest insertAirportTest = new
InsertAirportTest();
                        insertAirportTest.run(airports);
                        System.out.println "[" + new Date() + "] Insert
airports test done, run: " + (repetitions + 1) + ", database: " + testing_database + ", data
amount: " + toload);

                        InsertFlightTest insertFlightTest = new
InsertFlightTest();

                        LinkedList<Flight> flights = Util.loadFlights(new
File("1987.csv"), toload);
                        insertFlightTest.run(flights);
                        System.out.println "[" + new Date() + "] Insert
flights test done, run: " + (repetitions + 1) + ", database: " + testing_database + ", data amount:
" + toload);

                        SelectFlightTest selectFlightTest = new
SelectFlightTest();
                        selectFlightTest.run(flights);
```

```

                                System.out.println "[" + new Date() + "] Select
flights test done, run: " + (repetitions + 1) + ", database: " + testing_database + ", data amount:
" + toload);

                                ListFlightTest    listFlightTest    =    new
ListFlightTest();

                                listFlightTest.run(airports);
                                System.out.println "[" + new Date() + "] List
flights test done, run: " + (repetitions + 1) + ", database: " + testing_database + ", data amount:
" + toload);

                                HttpClient.sendGET(url + "/purge");

                                }

                                }

                                }

                                ConnectionPool.close();

                                } catch (Exception ex) {
                                    ex.printStackTrace();
                                }

                                }

}

```

## Appendix 31

File: \$root/server/app/controllers/Purge.java

```
package controllers;

import models.database.DB4o;
import models.database.MySQL;
import play.mvc.Controller;
import play.mvc.Result;

public class Purge extends Controller {

    public static Result purge() {
        MySQL mySQL = new MySQL();
        mySQL.purge();

        DB4o db4o = new DB4o();
        db4o.purge();
        return ok("[1]");
    }
}
```

## Appendix 32

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';
```

```
-- -----
-- Table `tests`
-- -----
```

```
CREATE TABLE IF NOT EXISTS `tests` (
  `test_id` INT NOT NULL AUTO_INCREMENT ,
  `date` TIMESTAMP NULL ,
  `type` INT NULL ,
  `database_type` INT NULL ,
  PRIMARY KEY (`test_id`) )
ENGINE = InnoDB;
```

```
-- -----
-- Table `responses`
-- -----
```

```
CREATE TABLE IF NOT EXISTS `responses` (
  `response_id` INT NOT NULL AUTO_INCREMENT ,
  `test_id` INT NULL ,
  PRIMARY KEY (`response_id`) ,
  INDEX `test_id_idx` (`test_id` ASC) ,
  CONSTRAINT `test_id`
    FOREIGN KEY (`test_id` )
    REFERENCES `tests` (`test_id` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

```
-- -----
-- Table `responses_parts`
-- -----
```

```
CREATE TABLE IF NOT EXISTS `responses_parts` (
  `responses_part_id` INT NOT NULL AUTO_INCREMENT ,
  `response_id` INT NULL ,
  `array_index` INT NULL ,
  `timevalue` MEDIUMTEXT NULL ,
  PRIMARY KEY (`responses_part_id`) ,
  INDEX `response_id_idx` (`response_id` ASC) ,
  CONSTRAINT `response_id`
    FOREIGN KEY (`response_id` )
    REFERENCES `responses` (`response_id` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

```
SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

## Appendix 33

File: \$root/result/net/pixomania/dbtest/result/database/Database.java

```
package net.pixomania.dbtest.result.database;

import java.io.*;
import java.sql.*;

public class Database {
    private Connection conn;
    private int selectmax;

    public Database(int selectmax){
        this.selectmax = selectmax;
        try {
            conn = DriverManager.getConnection("jdbc:mysql://alternia:3306/result",
"root", "");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public long calculateRequestTotal(int testId) {
        try {
            PreparedStatement pst = conn.prepareStatement("SELECT timevalue,
array_index FROM responses, responses_parts WHERE responses.test_id = ? AND
responses_parts.response_id = responses.response_id");
            pst.setInt(1, testId);
            ResultSet rs = pst.executeQuery();
            long total = 0;
            long start = 0;
            int i = 0;
            while(rs.next()){
                if(rs.getInt(2) == 0) {
                    start = rs.getLong(1);
                } else if (rs.getInt(2) == 4) {
                    total += rs.getLong(1) - start;
                    i++;
                    if (i == this.selectmax) break;
                }
            }
            return total;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return -1;
    }

    public void scatter(int testId) {
        File f = new File("output.txt");
        try {
            BufferedWriter bw = new BufferedWriter(new FileWriter(f));
            PreparedStatement pst = conn.prepareStatement("SELECT timevalue,
array_index FROM responses, responses_parts WHERE responses.test_id = ? AND
responses_parts.response_id = responses.response_id");
            pst.setInt(1, testId);
            ResultSet rs = pst.executeQuery();
            long total = 0;
            long start = 0;
            int i = 0;
            while (rs.next()) {
                if (rs.getInt(2) == 2) {
                    start = rs.getLong(1);
                } else if (rs.getInt(2) == 3) {
```



```

        bw.write(i + "\t" + (rs.getLong(1) - start) + "\n");
        i++;
        if (i == this.selectmax) break;
    }
}

bw.close();

} catch (SQLException e) {
    e.printStackTrace();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}
}

public long calculateDatabaseTotal(int testId) {
    try {
        PreparedStatement pst = conn.prepareStatement("SELECT timevalue,
array_index FROM responses, responses_parts WHERE responses.test_id = ? AND
responses_parts.response_id = responses.response_id");
        pst.setInt(1, testId);
        ResultSet rs = pst.executeQuery();
        long total = 0;
        long start = 0;
        int i = 0;
        while (rs.next()) {
            if (rs.getInt(2) == 2) {
                start = rs.getLong(1);
            } else if (rs.getInt(2) == 3) {
                total += rs.getLong(1) - start;
                i++;
                if (i == this.selectmax) break;
            }
        }
        return total;
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return -1;
}

public double calculateAverageRequest(int testId) {
    long total = calculateRequestTotal(testId);
    try {
        PreparedStatement pst = conn.prepareStatement("SELECT count(*) FROM
responses WHERE responses.test_id = ?");
        pst.setInt(1, testId);
        ResultSet rs = pst.executeQuery();
        rs.next();
        return ((double)total/rs.getDouble(1));
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return -1;
}

public double calculateAverageDatabase(int testId) {
    long total = calculateDatabaseTotal(testId);
    try {
        PreparedStatement pst = conn.prepareStatement("SELECT count(*) FROM
responses WHERE responses.test_id = ?");
        pst.setInt(1, testId);

```

```

        ResultSet rs = pst.executeQuery();
        rs.next();
        return ((double) total / rs.getDouble(1));
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return -1;
}

public double calculateRequestTotalAverage(int testId1, int testId2, int testId3) {
    long total1 = calculateRequestTotal(testId1);
    long total2 = calculateRequestTotal(testId2);
    long total3 = calculateRequestTotal(testId3);

    long total = total1 + total2 + total3;
    return ((double)total/3.0);
}

public double calculateDatabaseTotalAverage(int testId1, int testId2, int testId3) {
    long total1 = calculateDatabaseTotal(testId1);
    long total2 = calculateDatabaseTotal(testId2);
    long total3 = calculateDatabaseTotal(testId3);

    long total = total1 + total2 + total3;
    return ((double) total / 3.0);
}

public double calculateAverageRequestAverage(int testId1, int testId2, int testId3) {
    double total1 = calculateAverageRequest(testId1);
    double total2 = calculateAverageRequest(testId2);
    double total3 = calculateAverageRequest(testId3);

    double total = total1 + total2 + total3;
    return (total / 3.0);
}

public double calculateAverageDatabaseAverage(int testId1, int testId2, int testId3) {
    double total1 = calculateAverageDatabase(testId1);
    double total2 = calculateAverageDatabase(testId2);
    double total3 = calculateAverageDatabase(testId3);

    double total = total1 + total2 + total3;
    return (total / 3.0);
}
}

```