UNIVERSITÉ LIBRE DE BRUXELLES ULB

INFO-H-415 - Advanced Database
Object Oriented Database
DB4O

Anastasiia Zavolozhina 000454967
Ferdiansyah Dolot 000455509

December 2017

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Why OODBMS?

Object-oriented databases have been used since the late 1980s to provide database management for applications built in accordance with the concept of object-oriented programming. Object technology extends the traditional technique of application development with new data modeling and programming methods. To reuse code and improve the integrity of data in object programming, the data and code for their processing are organized into objects. Thus, restrictions on data types are almost completely removed.

If the data consists of short, simple fixed-length fields (name, address, bank account balance), then the best solution is to use a relational database. However, if the data contains *a nested structure, a dynamically variable size, user-defined arbitrary structures (e.g. multimedia)*, their presentation in a tabular form will be at least not easy. At the same time, in OODBMS, each user-defined structure is an **object, directly managed by the database**. In RDBMS communications are controlled by a user who creates foreign keys. Then, to detect links dynamically at runtime, the system scans two (or more) tables, comparing the foreign keys to match. This process, called a join, is the weak side of relational technology. More than two or three levels of associations - a signal to look for the best solution. In OODBMS, *the user simply declares a connection, and the DBMS automatically generates management methods, dynamically creating, deleting and traversing links.* Links are straightforward, there is no need to browse and compare or even search for an index that can greatly affect performance. Thus, the use of the object model is preferable for databases with a large number of complex links: cross references, links linking several objects to several (many-to-many relationships) bidirectional links.

Unlike relational, OODBMS fully *supports object-oriented programming languages.* Developers using C ++ or Smalltalk deal with one set of rules (allowing the use of object technology advantages such as inheritance, encapsulation, and polymorphism). The developer should not resort to translating the object model into a relational one and back. Application programs access and function with objects stored in a database that uses standard object-oriented language semantics and operations. In contrast, a relational database requires a developer to translate an object model to a supported data model and includes subroutines to ensure that this mapping is at runtime. The consequence is additional effort in developing and reducing efficiency.

Finally, OODBMS are *suitable* (without translations between the object and relational models) *for organizing distributed computing.* Traditional databases (including relational and some object databases) are built around a central server that performs all operations on the database. This architecture has a number of drawbacks and the main one is the scalability issue. Currently, workstations (clients) have a computing power of about 30 - 50 percent of the database server's power, that is, most of the computing resources are distributed among customers. Therefore, more and more applications, especially databases and decision-making tools, work in distributed environments in which objects (object program components) are distributed across many workstations and servers and where any user can access any object. Thanks to intercomponent interoperability standards (more on this later), all these code fragments are combined with each other regardless of hardware, software, operating systems, networks, compilers, programming languages, various querying and reporting tools, and dynamically change when objects are manipulated without loss of efficiency .

2 OODBMS

2.1 Overview

OODBMS (Object-Oriented Database Management System) is a DBMS in which information is represented as objects, their attributes, methods and classes , as in object-oriented programming languages. In a nutshell, the object-oriented approach is based on the following notions:

- *object and object identifier;*
- *methods and attributes;*
- *classes;*
- *class inheritance and hierarchy.*

Any real-world entity in object-oriented languages and systems is modeled as an **object**. Each object at its creation receives a unique **identifier** generated by the system, which is associated with this object all the time of its existence and does not change with the change of the state of the object. By the **s**tate of an object a set of values of its attribute is implied.
Each object also has it's **behavior**, which is a set of methods that operate on the object's state. The value of the attribute of the object is also some object or set of objects. The state and behavior of the object are encapsulated in the object. The interaction of objects is carried out on the basis of the transmission of messages and the implementation of appropriate methods.
A set of objects with the same set of attributes and methods forms an **object class**. An object belongs to only one class (if we do not consider inheritance). It is possible to have primitive predefined classes whose instance objects do not have attributes: integers, strings, etc. A class whose objects can serve as attribute values for objects of another class is called the **domain** of this attribute.
It is allowed to spawn a new class on the basis of an already existing class - **inheritance**. In this case, a new class, called a **subclass** of an existing class (**superclass**), inherits all attributes and methods of the superclass. In a subclass, in addition, additional attributes and methods can be defined.
There are cases of simple and multiple inheritance. In the first case, a subclass can be defined only on the basis of one superclass, in the second case there can be several superclasses. If the language or system supports single inheritance of classes, the set of classes forms a tree-like hierarchy. When maintaining multiple inheritance classes are connected in an oriented graph with a root, called the class lattice. A subclass object is considered to belong to any superclass of this class [4].
OODBMS technology implies the existence of an integrated language environment that simultaneously allows you to construct an object database that contains not only data, but also program code (object methods) that provides access to these data, and application code. Thus, the gap between passive data and active programs disappears, the project of the applied system is conducted within the framework of a single technology, which speeds up its development and facilitates subsequent support.

2.2 OODBMS advantages

Performance

Object-oriented database systems have several features that ensure their performance gains.

In the OODB, attribute's value of the 'X' object, whose domain is another 'Y' object is the identifier of the 'Y' object. Thus, if we need to extract 'Y' object after the 'X' object was already extracted, the DBMS can extract 'Y' by finding its identifier.

If the object's physical address is the identifier, then the object can be extracted directly. For the logical address (as the indentifier), the object will be found by the corresponding element of the hash table.

In the relation database model, this might not be so easy, because there is no support of object identifiers there. The second feature of OODB, that provides a performance growth, is that in most object-orieneted DBs when an object is loaded into memory, object identifiers stored in this object, are converted into memory pointers.

In RDBs identifiers are not stored. Thus, storing memory pointers to other tuples is also not possible. The absence of the ability to navigate through the objects contained in the memory is a fundamental property of the RDB. Resulting decrease of performance can not be compensated for simply by increasing the buffer memory capacity. So, when executing applications that involve multiple navigation on loaded linked objects, OODBs can significantly outperform the RDB in performance [7].

Defining Custom Abstractions

OODBs provide the ability of defining new abstractions and managing their implementation. They can correspond to data structures required in complex tasks, new abstract data types. In other words, with modern OODBs systems, user can develop a new class with it's attributes and methods, have classes inheriting attributes and methods from superclasses, create instances of the class, each of which has a unique object identifier, extract these instances one by one or by groups, and load and perform the methods. Moreover, OODBs let a user define objects as collections of other objects, and multiple nesting levels are allowed for collections. Properties can also be defined through the collection designer and have a complex structure. They can have non-primitive objects as values, which makes it possible to

form deeply nested object structures. When the RDB model expresses complex data structures through additional relationships and connections, OODB models use these multivalued properties. [LINK1]

Relations can be designed easier

In OODBs, a means of inverse relations provided to express mutual references between two objects (binary communication). Such a system provides referential integrity by establishing an appropriate backlink immediately after creating a direct link. There is even a possibility of automatic distribution of deletions via these links.

User-defined keys are not needed

In object-oriented database model, object identifier of each object is unique and automatically generated by the system. Therefore, OODB model eliminates the need of user-defined keys, which gives OODB model following advantages: an application can not modify the object identifier. Also, this feature implies a separate and coherent concept of identity, regardless of the way the object is being accessed or how it is modeled using descriptive data. Thus, if two objects have different object identifiers - they are not identical, even if the objects have the same structure, and the same values of all of their properties. In RDB model, where user-defined keys identify objects, such two objects would be considered the same object [6].

Less necessity of connections

A relational join is a mechanism that maps two relationships based on the values of the corresponding attribute pairs in these relations. Since in OODB two classes can have corresponding pairs of attributes, in this model, there still may be a need for a relational connection (or explicit connection). However, in comparison with RDBs, the need of connecting classes can be significantly reduced because of the support of the path expressions in OODBs. There are even situations in which the necessity of relational association can be completely eliminated.

Thus, in OODBs, implicit connections, generated by the hierarchical nesting of objects, differ from explicit associations. The latter resemble relational joins: two objects are explicitly compared by attribute values or object identifiers. Moreover, all explicit connections (based on comparing values or identifiers) can not be expressed in the relational query language, since in RDB only atomic attributes can be contained in any predicate.

Comparison predicates

For the relational database model, the comparison is always based only on values. In OODBs, if all key attributes of two tuples have the same values, those tuples are considered one entity. However, other types of comparison were developed and defined for the OODB model. Equality of of objects or properties can be compared based on their indentity or values.

Versioning and long transactions support

Support of versioning and long transactions is available in some OODBs, although with limited capabilities. RDBs do not support neither versioning, nor long transactions.

Object algebra

In comparison with relational algebra, object algebra is not as elaborated and developed. However, such an algebra exists, and it has fundamental operations: select, generate, map, union and difference. Other operations (e.g. intersection) can be defined based on these fundamental operations. 'Select' and 'Generate' operations produce a one-to-many mapping, while 'Map', 'Union' and 'Difference' mainly produce a one-to-one mapping. Saving objects means that algebraic operations do not create new objects and return objects belonging to previously defined database classes. 'Select' operator returns a subset of the input set. 'Union' operator returns objects that belong to a set A or a set B, or both. The 'Difference' operator returns objects contained in the set A, but not in B. 'Generate' operator generates objects from those that belong to the input set. 'Map' operator returns the set of objects formed as a result of each application of the method sequence.

2.3 OODBMS disadvantages

Controversial moments of technology. All OODBMSs by definition support the preservation and separation of objects. But, when it comes to practical application development on different OODBMS, there are a lot of differences in implementing support for three characteristics:

- *Integrity*
- *Scalability*
- *Fault tolerance*

OODBs do not require many of those internal functions and mechanisms that are so common and necessary in relational databases. For example, with a small number of users, long transactions and a minor server load, the object DBMS does not need to support complex backup / restore mechanisms. Nevertheless, the database technology is definitely mature for large projects.

To illustrate the first category, consider the mechanism for caching objects. As it was explained above,most object DBMS put the application code directly in the same address space where the DBMS itself is running. Therefore, performance improvement is often 10-100 times higher than with separate address spaces. But with this model, an object with an error can damage objects and destroy the database. There are two approaches to organizing a DBMS response to prevent data loss. Most systems send pointers to objects to the application, and sooner or later such pointers necessarily become incorrect. So, they are always wrong after moving an object to another user (e.g., after moving to another server). If the programmer develops the application is punctual, then there is no error. If the application attempts to use the pointer at the wrong time, then at the best there will be a system crash, at worst - information in the middle of another object will be lost and the integrity of the database will be violated [5].
There is a method better than using direct pointers. The DBMS adds an additional pointer and if necessary, if the object moves, the system can automatically resolve the situation (reload, if necessary, the object) without causing a conflict situation. There is another reason for applying indirect addressing: this allows tracking the frequency of object calls for organizing an effective swap mechanism. This is necessary to implement the second necessary property of databases - **scalability**. Again, mention should be made of the organization of distributed components. The classic client-server scheme, where the main load is on the client (this architecture is also called the "thick client-thin server"), copes better with this task than the mainframe structure, but it still can not be scaled to the enterprise level. Due to the multi-tier architecture of the client-server (N-Tier architecture), the computing load between the server and the end user is evenly distributed. The load is distributed over three or more links providing additional processing power. Thus, client-server architecture, which was recently considered a complex environment, gradually turned into an extremely complex environment because of the accelerated transition to the use of client-server systems of several links. A developer has to pay with additional complexity, time and problems associated with integration.

The third necessary quality of the database is fault tolerance. There are several ways to provide fault tolerance:

- *backup and restore;*
- *distribution of components;*
- *independence of components;*
- *copying.*

Following the first principle, a developer identifies potentially dangerous parts of the code and inserts into the program some actions that correspond to the beginning of the transaction - the storage of information necessary for recovery from a failure and the end of the transaction - the restoration or, if it is not possible, some other actions (e.g. sending a message to the admin). In modern DBMSs, this mechanism provides recovery in case of almost any error of the application, system or computer. Although, there is no ideal protection from failures.

In the mainframe architecture, the cause of failure was the central computer. When moving to a distributed multi-tier organization, errors can be caused not only by computers included in the network, but also by communication channels. In multi-tier architecture, if one of the links fails without special measures, the results of the work of others will be useless. Therefore, when developing distributed systems, a fundamentally higher level of fault tolerance is provided. Let's list the properties,that are mandatory for modern distributed DBMS:

• *transparent access to all objects regardless of their location*, so all DBMS services are available for the user and components can be redistributed without undue consequences.

• *the so-called "third-party transaction monitor"*, through which the transaction is executed in three stages - first, a request for readiness for the transaction is sent.

If one of the components fails,a system, created in accordance with the foregoing arguments, will suspend all users and interrupt all transactions. Therefore, the property of the DBMS, such as the independence of components, is important. With a network failure, the network is divided into parts, the components of each of which can not communicate with the components of the other part. In order to keep the possibility of working inside each such part, it is necessary to duplicate the critical information within each segment. Modern systems allow the database administrator to dynamically determine the network segments, thus varying the level of reliability of the entire system as a whole.

Finally, let's take a look on the copying (replication) of data. The simplest way is to add to each (main) backup server. After each operation, the main server transmits the changed data to the backup server, which is automatically turned on in the event of a major failure. Naturally, such a scheme is not without flaws. Firstly, this leads to significant overhead in duplicating data, which not only affects performance, but is in itself a potential source of failure. Secondly, in the event of a failure that caused a disconnection between the two servers, each of them will need to work in its segment of the network as the primary server, and the changes made on the servers during the time in this mode will not be synchronized even after restore the network. More perfect is the approach, when the necessary number of copies in the segment (selected in accordance with the required level of reliability) is created. This increases the availability of copies and even (with load sharing between servers) increases the speed of reading. The problem of the inability of updating data by several servers simultaneously in the case of their mutual inaccessibility is solved by allowing modifications in only one segment, e.g. having the largest number of users. With a well-tuned caching scheme, the overhead costs when duplicating the modified data are close to zero.

2.4 Comparison of OODBMS and RDBMS

In the previous chapters we listed and explained pros and cons of OODBMS and reviewed how those advantages and disadvantages are applicable to the relational model. Let's take a look at this comparison in a nutshel: • **Popularity.** There are many popular products, that are developed based on relational databases model. A lot of money were invested in the development and maintaining of these products and customers are ready to keep using and investing in it. In contrast, there are not many serious commercial products developed using OODBs and only few powerful OODBMSs exist.

• **The query language and its standardization.** Back in 1986, the first SQL-86 standard was set, which determined the fate of relational databases. After the setting of the standard, all relational DBMS developers should follow it. For object-oriented databases, there is no standard query language. Now, there is no consensus among developers about what and how this query language should should do.

• **Mathematical tools.** For the relational DB, there is the the foundation of the mathematical apparatus of relational algebra. This mathematical apparatus explains how basic operations on relations in the database should be performed, proves their optimality (or shows whether there is a need of optimization). On the other hand, there is no such tools for OODB. Thus, there are no strict terms in the OODB, such as the cartesian product, the relation, etc.

• **The problem of storing data and methods.** In relational databases only 'bare' data is stored. And what the application will do with this data depends on the application itself. In OODB, on the contrary, objects should be stored, and the object is a collection of its properties (object parameters) and methods (object interface). There is also no consensus on how OODBs should store objects and how the developer should develop and design these objects. Here, the problem of storing the hierarchy of objects, storing abstract classes, and so on appears.

OODB notation	RDB notation
Class	Relation
An instance of a class (object)	Tuple
Class Attribute	Column
Class hierarchy	Relationship hierarchy
Subclass	'Descendant' relation
Superclass	'Ancestor' relation
Methods	Data conversion rules

Table 1: OODB an RDB notations comparison

2.5 Model Overview

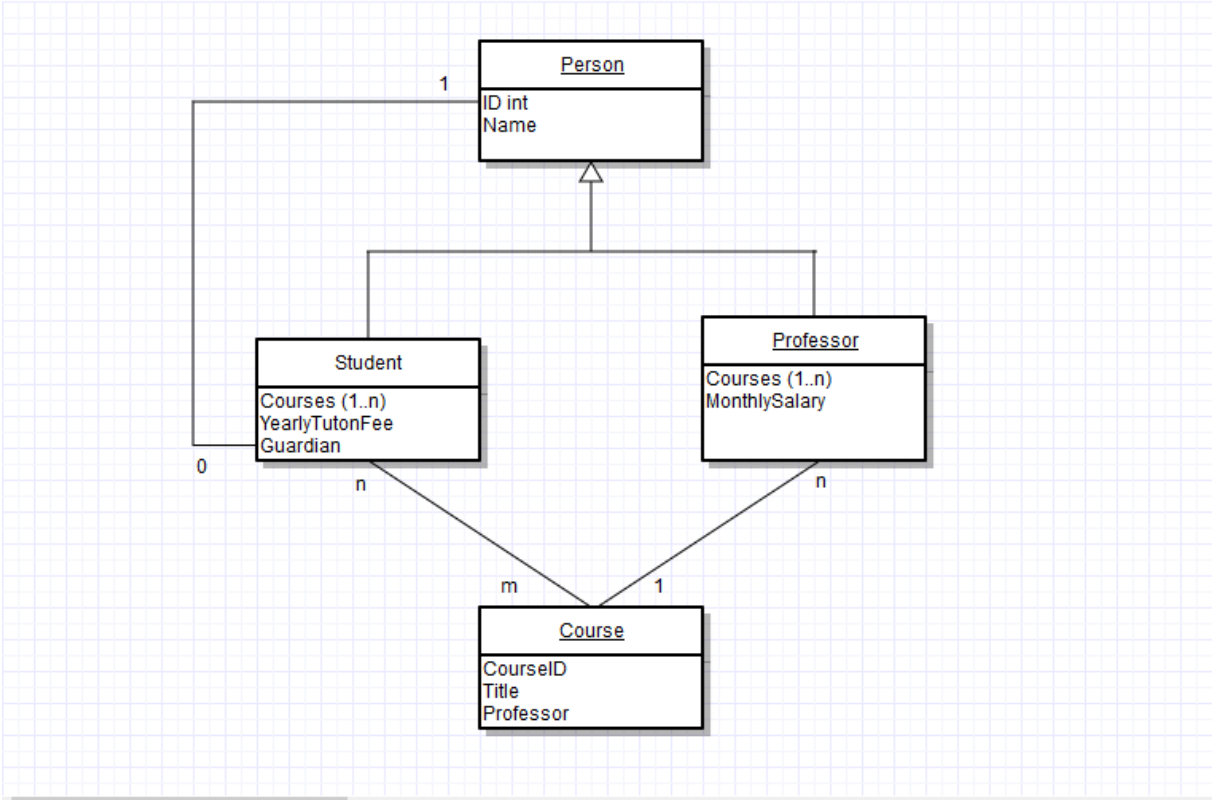


Figure 1: UML model

Figure 1 shows the UML model of our object-oriented database. Classes Student and Professor have a superclass Person. Person has attributes, that are common for both Student and Professor – ID and Name. Class Course contain courses with Title and Professor (who teaches the course) and both Professor and Student have attribute Course, which for Student is a list of courses he takes (one student can take many courses, course can be attended by many students) and for Professor is a list of courses he teaches (one professor can teach many courses, but there is only one professor per each course). Student has attribute YearlyTutionFee and Guardian (which is a Person object). One Student has one Guardian (as for the guardian, we don't store it's student to simplify the model).

2.6 Object Relationship

Links between objects in an object-oriented database are usually supported using class libraries. Each object whose information is stored in the OODB is considered to belong to a class, and the relationships between classes are established using the properties and methods of the classes.

In RDBMS communications are controlled by a user who creates foreign keys. Then, to detect links dynamically at runtime, the system scans two (or more) tables. In OODBMS, the user simply declares a

connection, and the DBMS automatically generates management methods, dynamically creating, deleting and traversing links, that are straightforward, there is no need to browse and compare or even search for an index that can greatly affect performance. Loading object-oriented data is complicated by the presence of relationships among the objects; these relationships prevent simply the use of a relational load utility for an OODB.

In RDB, all data stored in a tuple is either a string or a number and to referent to other tuples, it uses foreign keys, which are part of the user data. In ODB, objects use relationships to reference each other by their object IDs. IDs are created and maintained by the database itself and are usually not visible to the user (or are not available at all when the load files written, since the corresponding objects have not yet been created). Thus, relationships must be represented by some other means in the load file. This representation is called a surrogate identifier [10].
Relationships may be circular. An object X may refer to an object Y which refers back to X, either directly or by a chain of relationships. Therefore, the load utility y must be able to resolve surrogate identifiers to objects that have not yet been created when the surrogate is first seen.
Inverse relationship are relationships that are maintained in both directions, so that an update to one direction of the relationship causes an update to the other [10].

Suppose that object X has an inverse relationship with object Y. Then when Y's object identifier is stored in X, X's identifier should be stored in Y. The most obvious way to maintain inverse relationships (or, if each object is created separately, as by insert or new, this is the only way) is to update each inverse object immediately after realizing that the update is needed, e.g, updating Y immediately after creating X. There are two reasons why this method is not always appropriate: first, object Y may not have been created yet; second, this approach leads to performance several orders of magnitude worse than is possible using a different approach.

2.7 Inheritance

inheritance implies the ability to create from object classes new classes of objects that inherit the structure and methods of their ancestors, adding features that reflect their own individuality. Inheritance can be simple (one ancestor) and multiple (several ancestors); The mechanism by which class definitions extend to classes that lie below it in the hierarchy of class generalization. This allows you to repeatedly change the definitions, making the changes associated with the specialization as necessary. In our model (see Figure 1), inheritance is represented by superclass Person and it's subclasses Student and Professor.Student and Professor extend Person class, htey have their own specific attributes (YearlyTutionFee, Guardian, Salary, Courses), while ID and Name are general attributes for both and represented in Person.

2.8 Indexing

Indexing is an essential concept in database systems, that allows to speed up the evaluation of queries. Without using index structure, to evaluate a query the system checks for desired tuple by looking through the whole file. In relational database systems, indexes are especially useful when the user needs to select a small subset of a relation tuples based on the value of a specific attribute. In this case, the system looks up the desired attribute value in the index (stored in B-trees, or hash tables) and then retrieve the page that contains the desired tuples. Using index for searching influences the performance of producing the result but not the result itself.

In object-oriented database systems, indexing is much more complex than in relational ones. Objects, in contrast with relational tuples, are not flat. Thus, indexing should be possible on instance variables, nested several levels deep in an object that is being indexed.

For example, indexing on classes may cause authorization problems (allowing to build an index on one class can allow access to some unauthorized information). Let's suppose a Professor has access to Students who take the course he teaches, but not other Students, so if indexing is build on Student class, access authorization to certain student objects would be quite complicated [2].

2.9 Transaction

Transaction support allows developers to define the concept of a unit (quantum) of work. In database terminology, this means that the system must be able either to make a set of changes to the database immediately, or to ensure that none of these changes have been made. Within the transaction, either all operations are successful, or the transaction is not performed at all. In object-relational environments,

at a minimum, the commit and rollback operations must be supported. The development of an object-relational environment in a multi-user system can be complicated, and must be carefully thought out.

In addition to the storage environment, the application must be able to handle errors. If the transaction is interrupted or fails, the system should be able to return to a stable working state, for example, by reading the previous state information from the database. Therefore, the storage environment and the error handling environment must work in close interaction.

3 DB4O

3.1 Model Overview

We are currently using model as show on figure 1 to explain DB4O technologies.

```
1  package com.adb.model;
2  ..
3  // import
4
5  public class Person {
6      private int id;
7      private String name;
8
9      public Person(){
10         this.id = 0;
11         this.name = "";
12     }
13
14     public Person(int id){
15         this.id = id;
16     } LINK3
17
18     public Person(int id, String name){
19         this.id = id;
20         this.name = name;
21     }
22
23     ..
24     //Get Set methods
25 }
```

Listing 1: Person.java

```java
package com.adb.model;
..
// import

public class Student extends Person{
    private List<Course> courses;
    private int yearlyTuitionFee;

    public Student(){
        super();
        courses = new ArrayList<>();
        yearlyTuitionFee = 0;
    }

    public Student(int id, String name, List<Course> courses, int yearlyTuitionFee){
        super(id, name);
        this.courses = courses;
        for(Course course: courses){
            course.addStudent(this);
        }
        this.yearlyTuitionFee = yearlyTuitionFee;
    }

    public void addCourse(Course course){
        if(!courses.contains(course)) courses.add(course);
        if(!course.getStudents().contains(this)) course.addStudent(this);
    }
    ..
    // Get Set methods
}
```

Listing 2: Student.java

```java
package com.adb.model;
..
v// import

public class Professor extends Person {
    private List<Course> courses;
    private int monthlySalary;

    public Professor(){
        super();
        this.courses = new ArrayList<>();
    }

    public Professor(int id, String name, List<Course> courses, int monthlySalary){
        super(id, name);
        this.courses = courses;
        for(Course course: courses){
            course.setProfessor(this);
        }
        this.monthlySalary = monthlySalary;
    }

    public void addCourses(Course course){
        if(!courses.contains(course)) courses.add(course);
        course.setProfessor(this);
    }
    ..
    // Get Set methods
}
```

Listing 3: Professor.java

```java
package com.adb.model;

..
// import

public class Course {
    private int courseId;
    private String title;
    private Professor professor;
    private List<Student> students;

    public Course(){
    }

    public Course(int courseId, String title, Professor professor, List<Student> students){
        this.title = title;
        this.courseId = courseId;
        this.professor = professor;
        this.students = students;
        for(Student student: students){
            student.addCourse(this);
        }
    }

    public void addStudent(Student student){
        if(!students.contains(student)) students.add(student);
        if(!student.getCourses().contains(this))student.addCourse(this);
    }
    ..
    // Get Set methods
}
```

Listing 4: Course.java

Listing 1 to 4 shows the actual representation of the model in Java programming language. Within each class, object relationship is maintained in order to comply with referential integrity rules. The details will be explained on later section of this report.

3.2 Basic Operation

This section will provide several basic operation within Db4o database technology such as connecting to database, store object, update object, and delete object. During this section, we will introduce simple object selection method provided by db4o in order to verify our operation result (there are several way to do selection in db4o database. We will visit these kinds of object selection and query method for later section). Function *selectSimpleObject* take a class as the parameter, and find the corresponding class in database file *db/simpleobject*.

```java
public static void selectSimpleObject(Class c){
    ObjectContainer container = Db4o.openFile("db/simpleobject");
    ObjectSet objectSet = container.query(c);
    while(objectSet.hasNext()){
        System.out.println(objectSet.next());
    }
    container.close();
}
```

Listing 5: Selecting simple object

3.2.1 Connecting to Database

Listing 6 shows how to connect to db4o database. Db4o use file to store the object database (line 27), and perform the basic operation such as store, update, delete, and query to database using object container.

```java
package com.adb.database.connection;

..
// import

public class Db4oConnection {
    private static final String DBFILE= "db/dbfile" ;
    private ObjectContainer objectContainer;

    public Db4oConnection(){
        objectContainer = null;
    }

    public ObjectContainer getObjectContainer(){
        return objectContainer;
    }

    public void commit(){
        objectContainer.commit();
    }

    public void close(){
        objectContainer.close();
    }

    public void connect(){
        try{
            objectContainer = Db4o.openFile(DBFILE);
        }
        catch (Exception e){
            System.out.println(e.getMessage());
        }
    }

    public  void defrag() throws Exception{
        Defragment.defrag(new DefragmentConfig(DBFILE));
    }
}
```

Listing 6: Db4oConnection.java

3.2.2 Store

Listing 7 shows a snippet of code on how to store a person object in db4o database. It starts by opening a db4o file to get *ObjectContainer*, followed by a call of method store with the corresponding person object in the parameter. The container then commit in order to confirm that the transaction is done, then close the connection.

```java
public static void storeSimplePerson(){
    Person person = new Person(1, "Batman Robin");
    ObjectContainer container = Db4o.openFile("db/simpleobject");
    container.store(person);
    container.commit();
    container.close();
}
```

Listing 7: Store Person

Running of listing 5 selection object method will produce output as shown on listing **??**.

```
1, Batman Robin
```

Listing 8: Store person result

3.2.3 Update

Updating an object in db4o is simply by getting the existing object in db4o database we wish to update, store it in object reference variable, and do call of mutator method in the attribute we wish to update,

provided with the new value. Listing 9 shows a flow of updating existing person object in db4o. Selection result of updated object can be seen on listing 10.

```
public static void updateSimplePerson(){
    ObjectContainer container = Db4o.openFile("db/simpleobject");
    ObjectSet<Person> objectSet = container.query(Person.class);
    Person p = objectSet.next();
    p.setName("The Dark Knight");
    container.store(p);
    container.commit();
    container.close();
}
```

Listing 9: Update Person

```
1, The Dark Knight
```

Listing 10: Update person result

3.2.4 Delete

Delete operation in db4o is also using container access to db4o database file. We will first find the corresponding object by doing query, iterate result of the query, and use delete function in container, and commit to indicate the transaction has finished. Listing 11 shows the complete code on how to perform object deletion in db4o.

```
public static void deleteSimplePerson(){
    ObjectContainer container = Db4o.openFile("db/simpleobject");
    ObjectSet<Person> objectSet = container.query(Person.class);
    while(objectSet.hasNext()){
        container.delete(objectSet.next());
    }
    container.commit();
    container.close();
}
```

Listing 11: Delete Person

3.3 Querying Object

As mentioned previously, several type of queries/object selection methods exists in db4o database, such as Query by example (QBE), Native queries, and SODA (Simple Object Data Access) query.

3.3.1 Query by Example (QBE)

Query by example (QBE) uses object template to perform query in db4o object container database. Depending on how you set the attributes' value of your object, db4o will match existing object stored with template object except the default values like null (for object), zero (for integer), etc. Listing 12 shows how to do query by example in db4o.

```java
package com.adb.database.query;

..
// import

public class QBEQuery {
    private Db4oConnection db4oConnection;

    public QBEQuery(Db4oConnection db4oConnection){
        this.db4oConnection = db4oConnection;
    }

    public ObjectSet queryStudent(Student student){
        return db4oConnection.getObjectContainer().queryByExample(student);
    }
}
```

Listing 12: QBE sample

To test listing 12 snippet code, we use main class in listing 13. We created an object of student which has id equals to 1. We will then find the object in db4o database and output the result of corresponding student in object set of students. Result is shown in listing 14. From the way of constructing QBE, we know that it is quite simple way to get an object, but will not be a good case when it comes to query with specific logical rules.

```java
package com.adb;
..
// import

public class Main {
    private static Db4oConnection db4oConnection;
    public static void main(String args[]) throws Exception{
        Db4o.configure().activationDepth(1);
        db4oConnection = new Db4oConnection();
        db4oConnection.connect();

        QBEQuery qbeQuery = new QBEQuery(db4oConnection);
        Student student = new Student(1); // create student object with id = 1
        ObjectSet<Student> objectSet = qbeQuery.queryStudent(student);
        printObjectSet(objectSet);
        db4oConnection.close();
    }

    public static void printObjectSet(ObjectSet objectSet){
        while(objectSet.hasNext()){
            System.out.println(objectSet.next());
        }
    }
}
```

Listing 13: Main QBE

```
1, Xanthus Wagner, 6911, Cally Forbes
```

Listing 14: QBE result for student with id equals 1

3.3.2 Native Query (NQ)

Native query uses *Predicate* class to return object set of the corresponding class. It provides simplicity to write the queries in the way of the programming language (in this case, Java) works, e.g object's attribute comparison using equality.

```java
package com.adb.database.query;

..
// import

public class NativeQuery {
    private Db4oConnection db4oConnection;

    public NativeQuery(Db4oConnection db4oConnection){
        this.db4oConnection = db4oConnection;
    }

    public ObjectSet execute(Predicate predicate){
        return db4oConnection.getObjectContainer().query(predicate);
    }
}
```

Listing 15: NQ Sample

Listing 15 shows how db4o execute query based on predicate. Notice that native query requires predicate to search for the object. Sample predicate is provided in listing 16. Class *StudentId* need to extends class *Predicate* with generic type of the object, and override *match* method to indicate a condition about when the predicate will consider the object as a match. In this case, we consider the predicate to be matched if the student id is less than the parameter id that is passed to predicate constructor call.

```java
package com.adb.database.predicate;

..
// import

public class StudentId extends Predicate<Student>{
    private int id;

    public StudentId(int id){
        this.id = id;
    }
    public boolean match(Student student){
        return student.getId() < id;
    }
}
```

Listing 16: NQ Predicate

```java
package com.adb;
..
// import
package com.adb;

public class Main {
    private static Db4oConnection db4oConnection;
    public static void main(String args[]) throws Exception{
        Db4o.configure().activationDepth(1);
        db4oConnection = new Db4oConnection();
        db4oConnection.connect();

        NativeQuery nativeQuery = new NativeQuery(db4oConnection);
        ObjectSet<Student> objectSet = nativeQuery.execute(new StudentIdPredicate(4));
        printObjectSet(objectSet);
        db4oConnection.close();
    }

    public static void printObjectSet(ObjectSet objectSet){
        while(objectSet.hasNext()){
            System.out.println(objectSet.next());
        }
    }
}
```

Listing 17: Main NQ

```
1, Xanthus Wagner, 6911, Cally Forbes
2, Roanna Bates, 5178, Renee Knowles
3, Leroy Yates, 4720, Jameson Beasley
```

Listing 18: Native query result for student with id less than 4

Listing 17 provides the main class to test a simple native query and output the result as shown on listing 18.

3.3.3 Simple Object Data Access (SODA) Query

SODA query is also known as query graph. Each graph consists of several items defined as follows.

- **Node**: Class or attribute within class.
- **Edge**: Relationship defined between each node. In SODA query, this is also known as *descend*.
- **Constraint**: Evaluation performed to candidate object whether to include it or not.

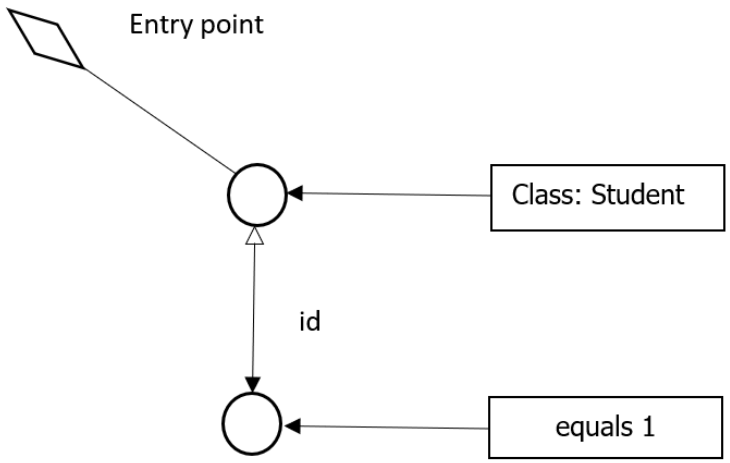


Figure 2: SODA query graph representation example

```java
package com.adb.database.query;

..
// import

public class SODAQuery {
    private Db4oConnection db4oConnection;
    private Query query;

    public SODAQuery(Db4oConnection db4oConnection){
        this.db4oConnection = db4oConnection;
        this.query = db4oConnection.getObjectContainer().query();
    }

    public ObjectSet queryStudentId(int id){
        query.constrain(Student.class);
        query.descend("id").constrain(id).equals();
        return query.execute();
    }
}
```

Listing 19: SODA Query

Sample graph representation is shown on figure 2. The graph starts at entry point, and define *Student* instance to be included in query result. We will then descend to *Student's id* and perform a match with id equals 1.

Listing 19 shows the identical implementation of the graph on figure 2. First constrain is introduced to the query object, which will limit the query evaluation to only the instance of *Student*. The *id* label indicates a *descent* or move to another node, in this case will be *id* of student instance. Then another constrain is introduced, which will only look for student's id equals to 1.

Table 2 and provide some important keywords of SODA query in db4o.

Keyword	Description
constrain	Attach a constrain to a node
descend	Move to one node as specified on descend parameter
orderAscending	Order the result of query ascending
orderDescending	Order the result of query descending

Table 2: SODA Query keyword

3.4 Object Relationship

Like any other database management system, db4o also has its own way of representing object relationship. This section will provide information about object inheritance, one to one, one to many, and many to many relationship between objects.

3.4.1 Inheritance

Inheritance is one of the most popular abstraction in object oriented programming. Like any other implementation of inheritance concept, db4o handles superclass, subclass, and interfaces the same way as they are implemented in object oriented programming language level, in this case we will be using Java.

```
1
2  public class Person {
3      private int id;
4      private String name;
5
6          ..
7      // Constructors
8
9          ..
10     // Methods
11 }
12
13 public class Student extends Person{
14     private List<Course> courses;
15     private int yearlyTuitionFee;
16     private Person guardian;
17
18     ..
19     // Constructors
20
21     ..
22     // Methods
23 }
24
25
26 public class Professor extends Person {
27     private List<Course> courses;
28     private int monthlySalary;
29
30     ..
31     // Constructors
32
33     ..
34     // Methods
35 }
```

Listing 20: Object inheritance

Listing 20 shows relationship between Student - Person, and Professor - Person. Being a super class, person shows more general level of student and professor (both has identification and name). On the other hand, student and professor have specific attribute that only attach to their instances. In this example, it is shown that student has list of courses to be attended, yearly tuition fee, and also guardian, meanwhile professor has list of courses to be taught, and monthly salary.

Other inheritance concept such as interfaces and abstract class can also be implemented in db4o the same way they are implemented in Java, hence on this report, we will limit our scope only in subclass - superclass implementation as shown in listing 20.

3.4.2 One-to-One Relationship

One to one relationship is defined when an entity has exactly one entity corresponded to it. To put a simple example, student object has one person object which is the guardian (as seen on listing 21). Normally, we would also store the student of each guardian as inverse one-to-one relation, but for the matter of simplicity, we will skip the part of inverse one-to-one relation for now. There will be samples of inverse relation on next following sections.

```
1  public class Student extends Person{
2      private List<Course> courses;
3      private int yearlyTuitionFee;
4      private Person guardian; // one person has one guardian
5
6      ..
7      // Constructors
8
9      ..
10     // Methods
11 }
```

Listing 21: One-to-One relationship example

3.4.3 One-to-Many Relationship

We have seen a lot of examples where one object which has relationship with a collections of object. In our model, relationship between professor and course is defined as one-to-many relationship. Listing 22 shows that for each professor object, it exists collections of courses. Meanwhile for course, there is an inverse relation with one professor. Please note that each time professor object is created during the constructor, code on line number 7 to 8 ensure that the object relationship is maintained between professor and course. It applies the same for course object with professor object each time the object is created (see listing 22 line 26).

```
1  public class Professor extends Person {
2      private List<Course> courses;
3      ..
4
5      public Professor(int id, String name, List<Course> courses, int monthlySalary){
6          ..
7          for(Course course: courses){
8              course.setProfessor(this);
9          }
10     }
11
12     public void addCourses(Course course){
13         if(!courses.contains(course)) courses.add(course);
14         course.setProfessor(this);
15     }
16
17         ..
18 }
19
20 public class Course {
21     private Professor professor;
22     ..
23
24     public Course(int courseId, String title, Professor professor, List<Student> students){
25         ..
26         this.professor = professor;
27     }
28
29     public void setProfessor(Professor professor) {
30         this.professor = professor;
31     }
32     ..
33 }
```

Listing 22: One-to-Many relationship example

3.4.4 Many-to-Many Relationship

The more complex form of entity relationship is many-to-many relationship. An example that is defined on our model is the relationship between student and course. Listing 23 shows that each time an instance of student is created, it also performs a call to *addStudent* method in class course, to make sure that the inverse relation and referential integrity exist between student and course. It also applies the same for course.

```
1  public class Student extends Person{
2      private List<Course> courses;
3      ..
4
5      public Student(int id, String name, List<Course> courses, int yearlyTuitionFee, Person
           guardian){
6          ..
7          for(Course course: courses){
8              course.addStudent(this);
9          }
10     }
11
12     public void addCourse(Course course){
13         if(!courses.contains(course)) courses.add(course);
14         if(!course.getStudents().contains(this)) course.addStudent(this);
15     }
16     ..
17 }
18
19
20 public class Course {
21     private List<Student> students;
22     ..
23
24     public Course(int courseId, String title, Professor professor, List<Student> students){
25         ..
26         for(Student student: students){
27             student.addCourse(this);
28         }
29     }
30
31     public void addStudent(Student student){
32         if(!students.contains(student)) students.add(student);
33         if(!student.getCourses().contains(this))student.addCourse(this);
34     }
35     ..
36 }
```

Listing 23: Many-to-Many relationship example

3.4.5 Inverse Relationship

We have seen inverse relationship example in one-to-many and many-to-many relationship before. The fact that inverse relationship can be useful in some situation does not make it a mandatory to be done. Let's find an example.

During our model description, it is important to know what courses do the students take, and also who are the students who take specified courses. If it turns out the information needs by the university is just what courses do the students take, hence we do not need to store inverse relation of students in courses although in relational database, we can not apply such rules.

3.5 Advanced Topic

Following section will explain several advance topic specifically related with db4o technology.

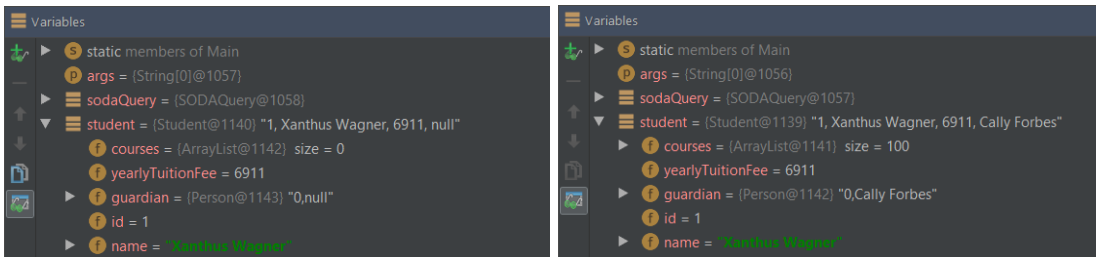
3.5.1 Object Activation

Object activation is very useful when it comes to querying object that has reference to other object. Revisiting previous section, we know that there is exist a relationship between student and person or student and course. Snippet code on listing 24 will demonstrate how object activation can be valuable for object retrieval.

```
1  public class Main {
2      private static Db4oConnection db4oConnection;
3      public static void main(String args[]) throws Exception{
4          Db4o.configure().activationDepth(1); // configure default activation depth
5
6          db4oConnection = new Db4oConnection();
7          db4oConnection.connect();
8
9          SODAQuery sodaQuery = new SODAQuery(db4oConnection);
10         Student student = (Student) sodaQuery.queryStudentId(1).next();
11         // increase activation depth
12         db4oConnection.getObjectContainer().activate(student, 2);
13
14         db4oConnection.commit();
15         db4oConnection.close();
16     }
17 }
```

Listing 24: Object activation example

Relationship between object can be quite large in more complex system, hence it is important to limit how much objects do we want to retrieve. On listing 24, we first set default activation to minimum value, which is one. This ensure that only primitive type of student's attributes are going to be retrieved. We then put a breakpoint on line 14 after student object activation is increased to two. Figure 3b shows that number of courses is 100 and person object (guardian) become exist, in contrast figure 3a shows no reference to the corresponding object (person and courses). By using this feature, we have flexibility to control how much amount of data we wish to get, and get a faster execution for object relationship type of queries.



(a) Before object activation (b) After object activation

Figure 3: Object activation example

3.5.2 Client-Server Architecture

Although up for now we only see standalone version of db4o, there is exist client-server architecture inside db4o technologies. Listing 25 show a very simple example of running client-server architecture in db4o. Server starts by opening db4o database file and specifying a port for clients' access. In order for the server instance to run continously, we put the server in a runnable class. Then each of client can connect to running server, giving that the client's access was already granted and registered inside the server, and clients are using the right authentication. This explains that db4o is also can do concurrent access to the database, just like any other database management system.

```
class RunDb4oServer implements Runnable{
    public ObjectServer server;

    @Override
    public void run() {
        // Create server
        server = Db4o.openServer("db/dbfile", 8732);
        db4oServer.server.grantAccess("user1", "pwd");
        db4oServer.server.grantAccess("user2", "pwd");
    }
}
public class Main {
    public static void main(String args[]) throws Exception{
        Db4o.configure().activationDepth(2);
        RunDb4oServer db4oServer = new RunDb4oServer();
        db4oServer.run();

        // Create clients
        ObjectContainer client1 = Db4o.openClient("interstellar", 8732, "user1", "pwd");
        ObjectContainer client2 = Db4o.openClient("interstellar", 8732, "user2", "pwd");


        Student student1 = client1.query(new StudentIdPredicate(1)).next();
        Student student2 = client2.query(new StudentIdPredicate(2)).next();
        System.out.println(student1);
        System.out.println(student2);

        client1.close();
        client2.close();
    }
}
```

Listing 25: Db4o client-server architecture

3.5.3 Transaction

These are two keywords that are related with transaction.

- **Commit**: Indication that transaction is finish.
- **Rollback**: There is problem when trying to modifying the database. Hence database state needs to be restored to the state before performing the transaction. This ensure consistency within db4o database.

Listing 27 shows a very simple way to handle undesirable transaction. We created an exception class (listing 26) to be thrown by *setYearlyTuitionFee* method in student (listing 28 when there is an attempt to modify student's tuition fee to negative amount. The test shows that the state of database has been rollback to previous state - no negative amount of yearly tuition fee (see listing 44 for the reference).

```
public class NegativeAmountException extends Exception {
    public NegativeAmountException(){
    }

    @Override
    public void printStackTrace() {
        super.printStackTrace();
    }
}
```

Listing 26: Exception class

```java
package com.adb;

..
// import

public class Main {
    private static Db4oConnection db4oConnection;
    public static void main(String args[]) throws Exception{
        Db4o.configure().activationDepth(2);

        db4oConnection = new Db4oConnection();
        db4oConnection.connect();

        Student student1 = new Student();
        try {
            System.out.println("Trying to change yearly tuition fee");
            System.out.println("--");
            student1 = db4oConnection.getObjectContainer()
                    .query(new StudentIdPredicate(1)).next();
            student1.setYearlyTuitionFee(-1000);
        }
        catch (NegativeAmountException e){
            e.printStackTrace();
            db4oConnection.getObjectContainer().rollback();
        }
        finally {
            db4oConnection.commit();
        }
        System.out.println("--");
        System.out.println("Value after trying perform the transaction");
        System.out.println(student1);

        db4oConnection.close();
    }
}
```

Listing 27: Db4o transaction

```java
public class Student extends Person{
    ..

    public void setYearlyTuitionFee(int yearlyTuitionFee) throws NegativeAmountException {
        if(yearlyTuitionFee < 0) throw new NegativeAmountException();
        this.yearlyTuitionFee = yearlyTuitionFee;
    }
}
```

Listing 28: Negative amount of tuition fee is not allowed

```
Trying to change yearly tuition fee
--
com.adb.exception.NegativeAmountException
        at com.adb.model.Student.setYearlyTuitionFee(Student.java:52)
        at com.adb.Main.main(Main.java:47)
Rollback transaction: Negative amount found
--
Value after trying perform the transaction
1, Xanthus Wagner, 6911, Cally Forbes
```

Listing 29: Running db4o transaction result

3.5.4 Refactor

In the practice, it is possible for a current schema to have changes. Just like in relational database management system, we could update the structure of our db4o database.

- **Adding an attribute**: Same with relational database, db4o will automatically add an attribute to old object returned from query result. This will not produce any error.

- **Renaming an attribute**: Renaming an attribute can be done either implicitly or explicitly. By doing implicitly, db4o will automatically delete the attribute that is not found in current class structure, and insert a new one. By doing explicitly, we need to run a db4o configuration in our application. Example for doing this is shown in listing 30.

```
1 Db4o.configure().objectClass("Student.class")
2     .objectField("guardian").rename("studentGuardian");
```

Listing 30: Renaming attribute in db4o

- **Changing an attribute type:** Db4o handles attribute changing by creating a new attribute and make the old value hidden from the query. However, if it is needed to use the old value within old data type, some data migration code needs to be implemented.
- **Removing an attribute**: If we remove an attribute from a class, db4o will automatically store the new object in database and ignore the deleted attribute.
- **Renaming a class**: Renaming class in db4o can be achieved by using command on listing 31.

```
1 Db4o.configure().objectClass(Student.class)
2     .rename(NewStudent.class.getName());
```

Listing 31: Renaming class in db4o

3.5.5 Indexing

It is common for any database management system to have indexes, so does db4o. Listing 32 shows how to create index in student id attribute. Please note that any refactoring process related with indexed field will cause the index to be recreated.

```
1 Db4o.configure().objectClass(Student.class)
2     .objectField("id").indexed(true);
```

Listing 32: Indexing in db4o

3.5.6 Defragment

Defragment is one of optimization in db4o in terms of file size. Let's say we decided to clear our database and start to recreate content inside it. We then perform object deletion for any object stored. After doing such and see the file size, it is clearly seen that the file size is not reducing, in contrast deleting will cause a file become bigger. By using defragment, we eliminate such problem. Snippet code on listing 33 shows a simple command to do defragment for database file. Db4o then create a backup database file named "currentdatabasefilename.backup" and remove unnecessary storage from current database file so that we geta nearly empty size of database file.

```
1  package com.adb.database.connection;
2
3  ..
4  // import
5
6  public class Db4oConnection {
7      private static final String DBFILE= "db/dbfile" ;
8
9      ..
10     // Methods
11
12     public  void defrag() throws Exception{
13         Defragment.defrag(new DefragmentConfig(DBFILE));
14     }
15 }
```

Listing 33: Defragment in db4o

4 Benchmarking

Db4o brings object oriented approach in storing and manipulating data, in contrast with table structure like relational database. However, we are interested to see how db4o's performance with regards to relational database. In our case, we choose postgresql database to perform schema mapping between currently implemented object oriented structure to RDBMS table, and perform query comparison within several scenarios.

4.1 Scenario

4.1.1 Database Precondition

Table 3 shows data precondition before executing queries in db4o and postgresql.

	Student	**Course**	**Professor**	**Course Taken**
Set-1	1000	100	100	100000
Set-2	5000	100	100	500000
Set-3	10000	100	100	1000000
Set-4	10000	500	500	5000000
Set-5	20000	500	500	10000000

Table 3: Benchmarking: Number of records loaded in database

4.1.2 Benchmarking Methods

Several csv files are available with regards number of record provided in table 3. Step of benchmarking is described as follow.

- **Database Server** In order to get more accurate result, we use db4o client-server architecture to have more similar result with postgresql which is also running on client-server mode.
- **Data Loading** First, we the mapping between our object oriented model to relational model. Then, we use psql command to load csv files into postgresql tables. For db4o, we read the csv input files and construct objects with its relation by using Java programming language.
- **Benchmarking Process** We developed several queries that have some meaning, by using db4o queries and postgresql queries. Please note that we will run postgresql queries using JDBC. We will do six iteration for every queries, and don't take the first run into account. So in total, we will take average of five queries running.

4.2 Queries

4.2.1 Simple Selection

Simple selection queries for db4o and postgresql can be seen on listing 34 and 35. The queries will select students with id equals to 111.

```
1 public ObjectSet simpleSelect(){
2     Query query = client.query();
3     query.constrain(Student.class);
4     query.descend("id").constrain(111).equal();
5     return query.execute();
6 }
```

Listing 34: Db4o simple selection query

```
1 public static ResultSet simpleSelect() throws SQLException{
2     String query = "SELECT * FROM student WHERE studentid = 111";
3     return execute(query);
4 }
```

Listing 35: PostgreSQL-JDBC simple selection query

4.2.2 Complex Selection

Listing 36 and 37 show equal query of selecting students who have yearlyTuitionFee greater than 1000 and name starts with 'N'.

```
public ObjectSet complexSelect(){
    Query query = client.query();
    query.constrain(Student.class);
    query.descend("yearlyTuitionFee").constrain(1000).greater();
    query.descend("name").constrain("N").startsWith(true);
    return query.execute();
}
```

Listing 36: Db4o complex selection query

```
public static ResultSet complexSelect() throws SQLException{
    String query = "SELECT * FROM student WHERE yearlytuitionfee > 1000" +
            "AND name like 'N%'";
    return execute(query);
}
```

Listing 37: PostgreSQL-JDBC complex selection query

4.2.3 Grouping and Aggregate Function

Grouping and aggregating in db4o is slightly different than relational database query like postgresql. Since db4o is operating on programming language operational level, there is no native way to do grouping other than doing it programmatically. The query is intended to get total of tuition fee each guardian holds. In Java, we are using map to group the object and sum the total of tuition fee, meanwhile in sql we can just specify the aggregate function and choose guardian name as the grouping column. Listing 38 and 39 show the complete code.

```
public Map<String, Integer> groupAndAggregate(){
    Query query = client.query();
    query.constrain(Student.class);
    ObjectSet<Student> objectSet = query.execute();
    Map<String, Integer> map = new HashMap<>();
    for(Student student: objectSet){
        if(map.containsKey(student.getGuardian().getName())){
            map.put(student.getGuardian().getName(), map.get(student.getGuardian().getName
                ()) + student.getYearlyTuitionFee());
        }
        else{
            map.put(student.getGuardian().getName(), student.getYearlyTuitionFee());
        }
    }
    return map;
}
```

Listing 38: Db4o aggregation query

```
public static ResultSet groupAndAggregate() throws SQLException{
    String query = "SELECT guardian, SUM(yearlytuitionfee) FROM student GROUP BY guardian";
    return execute(query);
}
```

Listing 39: PostgreSQL-JDBC aggregation query

4.2.4 Simple Join

Things are getting interested with join queries. Join expression in relational database is known as one of the most expensive operation to be done. In contrast, OODBMS provides a very simple way to perform join. If we recall from previous section, the link from one object to another is kept. Then the problem is simply iterating through object relation from one to another. Bear in mind that we need to use proper object activation depth in order to get the intended values. In this case, in order to get all record from

student joined with course taken table, we should have three depth of object activation. Listing 40 and 41 show the snippet code of the query.

```
Db4o.configure().activationDepth(3); // set activation depth to 3

public ObjectSet simpleJoin(){
    Query query = client.query();
    query.constrain(Student.class);
    return query.execute();
}
```

Listing 40: Db4o simple join query

```
public static ResultSet simpleJoin() throws SQLException{
    String query = "SELECT * FROM student a " +
            "INNER JOIN coursetaken b ON a.studentid = b.studentid";
    return execute(query);
}
```

Listing 41: PostgreSQL-JDBC simple join query

4.2.5 Complex Join

It applies the same principle principle from simple join to complex join for db4o queries, it is about tracing object link and references. For this complex queries, we want to take all the students relation to professor. In db4o, it is simply just extending the object activation depth to four.

```
Db4o.configure().activationDepth(4); // set activation depth to 4

public ObjectSet simpleJoin(){
    Query query = client.query();
    query.constrain(Student.class);
    return query.execute();
}
```

Listing 42: Db4o complex join query

```
public static ResultSet complexJoin() throws SQLException{
    String query = "SELECT * FROM student a " +
            "INNER JOIN coursetaken b ON a.studentid = b.studentid " +
            "INNER JOIN course c ON b.courseid = c.courseid " +
            "INNER JOIN professor d ON d.profid = c.profid ";
    return execute(query);
}
```

Listing 43: PostgreSQL-JDBC complex join query

4.3 Performance Evaluation

4.3.1 Import Time

Table 4 and figure 4 show execution time for importing data into database. Note that db4o execution time is faster in every case, although it still need to construct the object prior inserting data to database.

	PostgreSQL	**Db4o**
Set-1	3.041213805	0.4237
Set-2	7.0173037	0.75806
Set-3	16.630	5.54788
Set-4	64.549949529	21.760791
Set-5	129.751606017	44.25873

Table 4: Benchmarking: Import Running Time in (s)

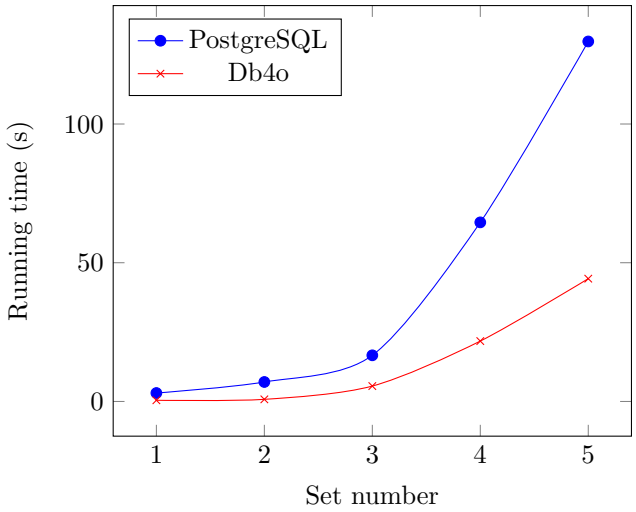


Figure 4: Benchmarking: Import running time differences

4.3.2 Simple Selection

Simple selection in PostgreSQL delivered quite consistent result from small to large table. This is expected because relational database provides internal optimization like table statistics to calculate the best query plan and executed it optimally. In the meantime, Db4o query have a quite slow performance compare to PostgreSQL query.

	PostgreSQL	**Db4o**
Set-1	0.00028	0.005103
Set-2	0.0003324	0.0072
Set-3	0.000383	0.0280287
Set-4	0.00036044	0.0112458
Set-5	0.00033921	0.04964

Table 5: Benchmarking: Import Running Time in (s)

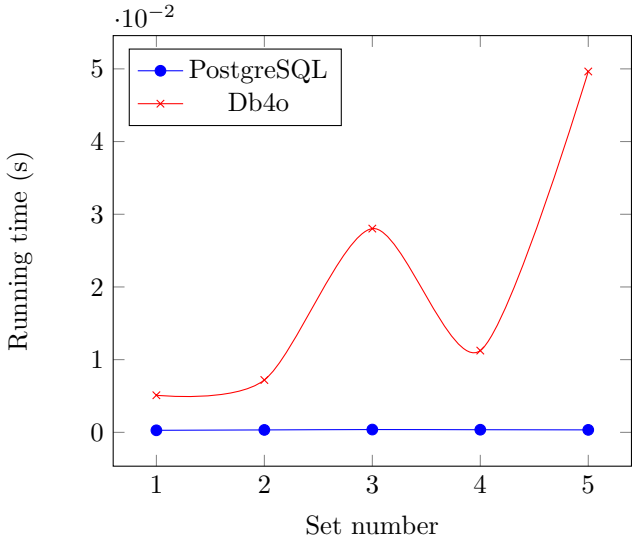


Figure 5: Benchmarking: Simple selection running time differences

4.3.3 Complex Selection

Complex selection derived the same behavior as simple selection and gives more or less the same pattern for runtime execution. Set-3 and Set-4 actually have the same number of record in student table, which the query was performed, but query in Set-3 tends to have slower result. Since we were running the result in host computer, it might be caused by some external resource that somehow use disk usage and increase the burden when doing the execution of Set-4. We would advice to rerun the benchmarking in cloud for later experiment.

	PostgreSQL	**Db4o**
Set-1	0.000380	0.004563
Set-2	0.000882	0.006940
Set-3	0.001424	0.035726
Set-4	0.001210	0.012431
Set-5	0.005672	0.045118

Table 6: Benchmarking: Complex Selection Running Time in (s)

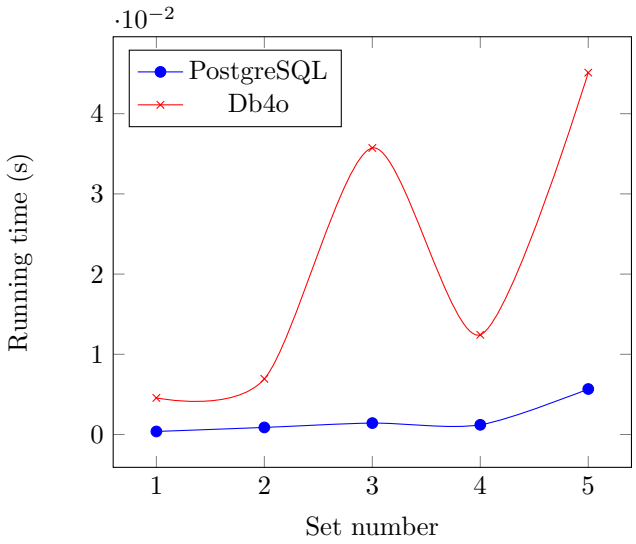


Figure 6: Benchmarking: Complex selection running time differences

4.3.4 Aggregation

As we have discussed on previous section, there is no native support for aggregation in db4o. Hence the process could be quite complicated, and the implementation is really depend on the programmer since it needs to be done manually on memory level. On the other hand, relational database management system has been developer for years using a lot of use cases, hence it is possible for it to do a lot of optimization, just like when doing complex queries. For that reasons, we think it is expected for Db4o to run significantly slower than PostgreSQL.

	PostgreSQL	**Db4o**
Set-1	0.000990	0.010666
Set-2	0.003606	0.016273
Set-3	0.006501	0.055613
Set-4	0.006466	0.229232
Set-5	0.012765	0.085981

Table 7: Benchmarking: Aggregation Running Time in (s)

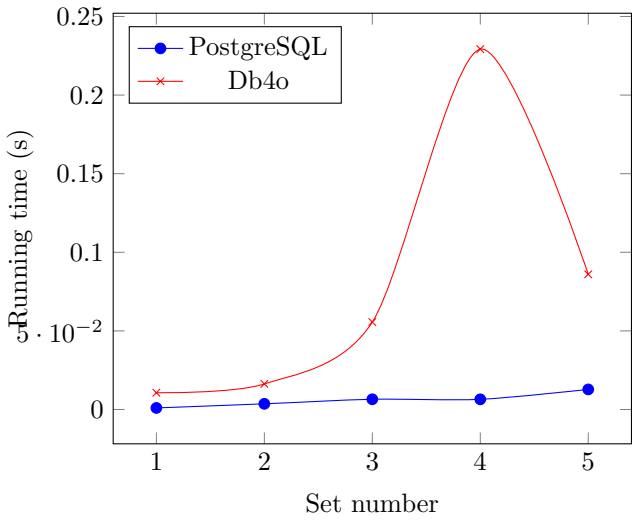


Figure 7: Benchmarking: Aggregation running time differences

4.3.5 Simple Join

In relational database, join operation is known as one of the most expensive operation. Meanwhile object oriented takes advantage from using its object relational link from one to another, hence all we have to do in db4o, as explained in previous section, is to set the activation depth into the level of relation that we wished to get. Result on figure 8 clearly explains this advantage.

	PostgreSQL	**Db4o**
Set-1	0.086618	0.000381
Set-2	0.347305	0.000710
Set-3	0.828559	0.000915
Set-4	7.331403	0.000599
Set-5	11.933769	0.001531

Table 8: Benchmarking: Simple Join Running Time in (s)

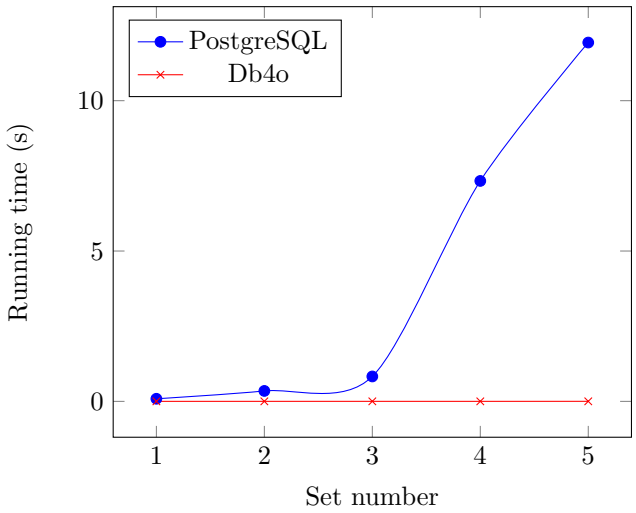


Figure 8: Benchmarking: Simple join running time differences

4.3.6 Complex Join

Same as simple join, complex join in db4o also takes benefit from object relation.

	PostgreSQL	Db4o
Set-1	0.132103	0.000436
Set-2	0.633576	0.000449
Set-3	1.980592	0.000933
Set-4	11.050936	0.000485
Set-5	RTE	0.001011

Table 9: Benchmarking: Complex Join Running Time in (s)

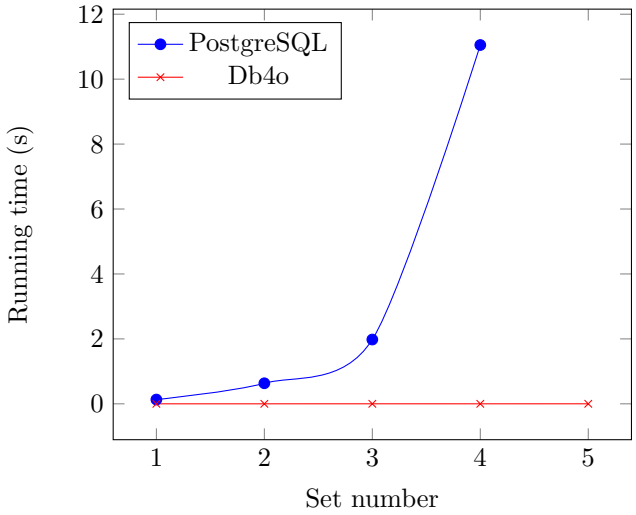


Figure 9: Benchmarking: Complex join running time differences

4.3.7 Adding Index

We tried to run the biggest working benchmark set with indexes in postgresql table on join attribute to see how significant the impact is.

Query	PostgreSQL	PostgreSQL with Index
Simple select	0.000360	0.000340
Complex select	0.001210	0.001609
Aggregation	0.006466	0.007206
Simple join	7.331403	6.835255
Complex join	11.050936	10.835177

Table 10: Benchmarking: Index and non index runtime comparison (s)

4.4 Running analyze and direct psql query

Notice that in our case, PostgreSQL was not able to completely run last set of complex query (see table 9 for reference). When we tried to look into the log generated, we found out that the problem was within java - out of memory.

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
        at java.util.jar.Attributes.read(Attributes.java:394)
        at java.util.jar.Manifest.read(Manifest.java:199)
        ...
        at org.postgresql.jdbc.PgStatement.executeQuery(PgStatement.java:231)
        at com.adb.database.query.JdbcQuery.execute(JdbcQuery.java:45)
        at com.adb.database.query.JdbcQuery.complexJoin(JdbcQuery.java:41)
        at com.adb.database.query.JdbcQueryExecutor.executeJdbcComplexJoin(
            JdbcQueryExecutor.java:40)
        at com.adb.benchmarking.JdbcBenchmark.benchmarkJdbcComplexJoin(JdbcBenchmark.java
            :62)
        at com.adb.benchmarking.BenchmarkingSet.start(BenchmarkingSet.java:85)
        at com.adb.benchmarking.MainBenchmarking.main(MainBenchmarking.java:30)
```

Listing 44: Java out of memory

Knowing that we have the problem on application side, we decided to take a look into analyze query and run the query directly into postgresql database equipped with index.

```
adb=# explain analyze SELECT * FROM student a INNER JOIN coursetaken b ON a.studentid = b.studentid INNER JOIN course c
ON b.courseid = c.courseid  INNER JOIN professor d ON d.profid = c.profid ;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
----------------
 Hash Join  (cost=1974.80..1217674.73 rows=25000120 width=84) (actual time=17.956..12735.840 rows=25000000 loops=1)
   Hash Cond: (b.courseid = c.courseid)
   ->  Hash Join  (cost=1938.00..873886.28 rows=25000120 width=42) (actual time=17.652..7945.362 rows=25000000 loops=1)
         Hash Cond: (b.studentid = a.studentid)
         ->  Seq Scan on coursetaken b  (cost=0.00..360621.20 rows=25000120 width=8) (actual time=0.007..1742.894 rows=2
5000000 loops=1)
         ->  Hash  (cost=922.00..922.00 rows=50000 width=34) (actual time=16.154..16.154 rows=50000 loops=1)
               Buckets: 65536  Batches: 2  Memory Usage: 2215kB
               ->  Seq Scan on student a  (cost=0.00..922.00 rows=50000 width=34) (actual time=0.005..4.716 rows=50000 l
oops=1)
   ->  Hash  (cost=30.55..30.55 rows=500 width=42) (actual time=0.296..0.296 rows=500 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 47kB
         ->  Hash Join  (cost=15.25..30.55 rows=500 width=42) (actual time=0.099..0.209 rows=500 loops=1)
               Hash Cond: (c.profid = d.profid)
               ->  Seq Scan on course c  (cost=0.00..9.00 rows=500 width=21) (actual time=0.011..0.033 rows=500 loops=1)
               ->  Hash  (cost=9.00..9.00 rows=500 width=21) (actual time=0.080..0.080 rows=500 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 35kB
                     ->  Seq Scan on professor d  (cost=0.00..9.00 rows=500 width=21) (actual time=0.007..0.034 rows=500
 loops=1)
 Planning time: 0.535 ms
 Execution time: 13223.512 ms
```

Figure 10: Explain analyze result of Set-5 complex join query

We can see from figure 10 in the first line of the result, it shows: *actual time=17.956..12735.840*. Based on postgresql documentation [9], actual time values are in milliseconds of real time, so that we know the query can take around 12000 milliseconds, or around 12 seconds to run. As the Set-4 running time for complex join query is around 11 seconds, this actual time is quite reasonable, although during the real time, depending also on the state of the disk, the result might be quite different.

5 Conclusion

Based on our research in db4o database, object relationship is one of the most significant advantages of db4o database compared to relational database. Object lineage is a powerful feature that will eliminate the join process happened in relational database, which considered as one of the most expensive operation.

However, there are also limitation in executing complex query like selection and aggregation. As we also know, relational database has a lot of internal performance optimization by using statistical analysis and perform it before query execution in order to reach the fastest execution. In contrast to that, db4o perform aggregation manually and complex operation can be painful for db4o.

We may say that some cases which has significant object lineage with limited processing of complex query may be useful to use db4o since it will eliminate modeling process, perform minimum part of configuration process, perform no-join operation and does not require external tool to be used.

6 References

[1] J. Stefan Edlich,H. Hörning, R. Hörning, 'The Definitive Guide to db4o. The first comprehensive guide to db4o, the open source native object database for .NET and Java', 2006

[2] D. Maier, 'Object-Oriented Database Theory An Introduction and Indexing in OODBS', TU Muenchen, 2001

[3] C. Marc, H. SchollQuery 'Query Optimization in an OODBMS', 2011

[4] C.S.R. Prabhu, 'Object-Oriented Database Systems: Approaches and Architectures, 2nd ed.', Prentice-Hall Of India Pvt. Limited, 2005

[5] S. Bagui, Achievements and Weaknesses of Object-Oriented Databases', Department of Computer Science, University of West Florida, U.S.A. ETH Zurich, Chair of Software Engineering © JOT, 2003

[6] I.Boyko, 'Object-oriented DBMS, 2000

[7] R. S. Ghongade, P J. Pursani, 'Comparison of Relational Database and Object Oriented Database', Scientific Journal Impact Factor (SJIF): 1.711 International Journal of Modern Trends in Engineering

[8] B. Nguyen, Object-oriented database programming with db4o, 12 Oct 2007 (Online)
https://www.codeproject.com/Articles/17946/Object-oriented-database-programming-with-db-o

[9] Using EXPLAIN
https://www.postgresql.org/docs/current/static/using-explain.html

[10] J.L. Wiener, J.F. Naughton, 'Bulk Loading into an OODB: A Performance Study', University of Wisconsin-Madison, 1994