Object databases and persistence

Promoteur de mémoire :
Pr. Esteban Zimányi
CoDE

Mémoire de fin d’étude
présenté par Willy-Dawid Khov
en vue de l’obtention du grade
d’Ingénieur Civil en Informatique
spécialisé en Ingénierie informatique
Abstract

In the world of object databases, the standards ODL and OQL were never adopted. So far, object database vendors have never developed a standardized query language and have preferred to have their own. But Microsoft has decided to step up and to propose its own vision of the query language: LINQ. The query capabilities are integrated into the programming language, which brings new possibilities for the developers. Moreover, its goal is to be able to query all types of databases. No need to learn dozens different syntaxes anymore. This is the query language that should be learned at school.
Acknowledgments

I would like to thank everyone who supports me during this year.

I would like to thank my promotor, Esteban Zimányi, for his support and his advices.

I would like to thank Boris Verhaegen and François Picalausa for their interest in my research.

Last but not least, I would like to thank my family who have always supported me, Julien Lamarre and Johnny Lucchese for their friendship, and most of all Tien-Ni Ung for her love.
# Contents

1 Introduction .................................................. 3

2 Object Database ............................................... 4
   2.1 Persistence .................................................. 6
   2.2 ACID Transaction Model ....................................... 7
   2.3 ODL and OQL .................................................. 7

3 Popular Object Databases ........................................ 8
   3.1 db4o .......................................................... 9
      3.1.1 Operations on Database ................................... 9
      3.1.2 OO Programming in Database ............................... 12
      3.1.3 Transactions .............................................. 13
      3.1.4 Collision ................................................ 14
      3.1.5 The Ways the Database Operates .......................... 14
      3.1.6 Replication ............................................... 16
      3.1.7 Schema Evolution ......................................... 17
      3.1.8 Callbacks ............................................... 18
   3.2 Versant Object Database ...................................... 19
      3.2.1 Architecture ............................................... 19
      3.2.2 Java Versant Interface ................................... 21
      3.2.3 Development steps ....................................... 22
      3.2.4 Persistence .............................................. 22
      3.2.5 Transaction ............................................... 23
      3.2.6 Object Modifications in Database ......................... 24
      3.2.7 Supported Collections ................................... 25
      3.2.8 Versant Query Language .................................. 26
      3.2.9 Events .................................................. 26
      3.2.10 Schema Evolution ........................................ 27
   3.3 Progress ObjectStore ......................................... 29
      3.3.1 Virtual Memory Mapping ................................... 29
      3.3.2 Architecture ............................................... 29
      3.3.3 Page Fetching ............................................ 31
      3.3.4 Page Permits and Locks .................................. 32
      3.3.5 Distribution and Heterogeneity ............................ 34
      3.3.6 Persistence ............................................... 34
      3.3.7 Database Layout ......................................... 35

1
Chapter 1

Introduction

The Object Oriented Database Management System (OODBMS) has been existing for 2 decades. At the Université Libre de Bruxelles, a part of a course is dedicated to the object databases. **ODL** and **OQL** which are the standards defined by ODMG are still learned. However, they are standards *on paper*. They are not supported by any database and so, the exercises are done also on paper.

This thesis will question the pertinence to learn about object databases today. Are OODBMS needed? If so, what is the query language that should be learned?

1. The object database and its characteristics will be presented. It will then be compared to the relational database.

2. Several object databases will be reviewed. Their architecture, their functioning and their query language will be explained.

3. Different potential standards for a query language will be studied to select the one that should be learned. Each one will have their syntax and possibilities exposed.

4. A database supporting the chosen standard must be selected so that exercises can be done on computer.

The second part (much tinier) of this thesis concerns the developpement of a LINQ-like for the programming language Java.
Chapter 2

Object Database

An object database is simply a database in which data are represented in the form of objects like in object-oriented programming. The RDBMS was and still is the most used in the world then why was there a need to develop a new type of database? This is because of the **Object-Relational Impedance Mismatch**.

In relational databases, the data is organized as tuples in relations. The model is very simple. So the representation of the real world objects is poor. There is a lack of semantic modelling. The data structure is homogeneous, there cannot be any complex type. Moreover, it is difficult to handle recursive and the many-to-many relationships are difficult to express.

Nowadays, applications are generally developped using object oriented programming language. The relational data model and the object model are two incompatible models. There must be a step of translation between the objects manipulated by the application and the data stored in the database. In fact, a lot of code and a lot of performance is allocated only for this conversion.

The role of the object databases was to unify the concepts used in the application and the database, and so, to avoid the **Object-Relational Impedance Mismatch**. The data are treated as objects which implies their identities, their attributes and their methods and their relationships with other objects. The hierarchy is extensible (inheritance, overloading and overriding, user-defined types). It supports schema evolution and long transactions. The performance is also improved. The OODBMS combines features from object-oriented programming language and database.

Unfortunately, so far, object databases lack a universal data model, are rather complex and above all, there is no standard supported for querying.

The object-oriented database manifesto defines 13 features that the database must have:

1. Complex objects.
2. Object identity.
3. Encapsulation.
4. Types and classes.
5. Type and class hierarchies.
6. Overriding, overloading and late binding.

7. Computational completeness.

8. Extensibility.


10. Efficiency.

11. Concurrency.

12. Reliability.


The approach taken by the relational database vendors was to add object-oriented concepts to their database: Object-Relational Database Management System (ORDBMS). The features added are:

- User-extensible type system.
- Encapsulation.
- Inheritance.
- Polymorphism.
- Dynamic binding of methods.
- Complex objects.
- Object identity.

Nevertheless, it clearly increases the complexity of the database and the mismatch between object-oriented application and the database still exists. Also, object-relational mapper can be used to automatize the conversion from the object-oriented model to the relational one.

The table 2.1 summarizes the main differences between the three types of database.

The object databases are not dead. There are still companies developing them and they do because there is a request. Many applications need to store complex data, a high performance object management, relationship navigation and transaction models for both long and short transactions. They are used in various domains such as: telecommunications, defense, financial services, transportation, bioinformatics, multimedia, ...
Table 2.1: Databases comparison.

Source: [35]

### 2.1 Persistence

The object persistence can be implemented differently. There are three big categories used in the object databases:

**Persistence by inheritance** Objects are made persistent if they inherit from a persistent class.

**Persistence by instantiation** Objects are made persistent at the instantiation.

**Persistence by reachability** Objects are made persistent if they can be reached by other persistent objects.
2.2 ACID Transaction Model

A transaction is a pack of operations on the database (reading, writing, ... ) which must respect the ACID properties. The interactions with the database are always done inside a transaction.

Atomicity A transaction is executed completely or not at all.

Consistency The data consistency must be preserved whatever happens.

Isolation The data manipulated by a transaction cannot be accessed by other transactions until it is terminated.

Durability Updates to persistent data cannot disappear even if there is a crash.

2.3 ODL and OQL

Object Definition Language and Object Query Language are the standards introduced by ODMG\textsuperscript{1} to increase the portability between the different object databases. The intention was good but they have never been adopted. Nowadays, there are not a single database which supports them.

\footnote{For more details, see \cite{36}}
Chapter 3

Popular Object Databases[13]

Several object database vendors are present on the market. Some of their products will be discussed to determine the specificities of each solution. Their way to handle object persistence and the possibilities offered to query objects in the database.

- **Objectivity/DB**
  It is an object-oriented database management system developed since 1993. The core is in C++ and there are different languages supported: C++, C#, Java, SmallTalk, Python, SQL++, and XML. Moreover, many platforms in 32bit or 64bit are supported such as Windows and Linux. Using cloud computing is also possible.

- **Progress ObjectStore**
  The languages supported are C++ and Java. All the .NET languages can be used too. There are two editions: the Personal Storage Edition Pro and the Enterprise. The first will be focused on. It is a lightweight object database with support for large but single-user databases. It also has a small memory footprint and is multithreaded. It is used in embedded systems, mobile computing and desktop applications.

- **Versant Object Database**
  Versant are specialized in developing object database management systems which are highly scalable and are based on a distributed object-oriented architecture. They have also created their own caching algorithm. The Versant Object Database which supports C, C++, Java and .NET is the market leader and is equipped with a lot of different tools.

- **Versant db4o**
  It is an open source object database which supports Java and .NET.

All the examples of code are taken or modified from [13].
3.1 db4o

In db4o, the database are represented as object container. It supports a local mode and a client/server mode. It contains one transaction. The next transaction is started when the previous one has finished. The links between stored and instantiated objects are maintained by the object container. It is in charge of the loading, updating and unloading of objects. Therefore, it is opened when the program needs the objects contained.

3.1.1 Operations on Database

All the operations are done through the object container.
To store objects, the method `store` of `ObjectContainer` is to be called. Any kind of objects can be stored. The persistence is by reachability.

```java
// create a new publication
Publication article = new Publication("Concepts for Object Databases");
// create authors
Author paul = new Author("Paul Jean");
Author michel = new Author("Michel Jean");
// add authors to the publication
article.addAuthor(paul);
article.addAuthor(michel);
// get the object container and save the article
ObjectContainer db = Db4oEmbedded.openFile("test.db");
db.store(article);
```

To retrieve objects, a query language language is needed. In db4o, there are three of them:

- Query by Example
- Native Queries
- SODA Queries

The first language uses the notion of prototype objects, so only objects that corresponds to the prototype can be retrieved. The method is `queryByExample`.

```java
ObjectContainer db = Db4oEmbedded.openFile("test.db");
// get author "Moira C. Norrie"
Author proto = new Author("Moira C. Norrie");
ObjectSet<Author> authors = db.queryByExample(proto);
for (Author author: authors) {
    System.out.println(author.getName());
}
// get all publications
ObjectSet<Publication> publications = db.query(Publication.class);
```
for (Publication publication: publications) {
    System.out.println(publication.getTitle());
}

The Native Queries are defined in the programming language. In this case, there is a type control. These queries are in fact transformed to SODA queries.

ObjectContainer db = Db4oEmbedded.openFile("test.db");
// find all publications after 1995
ObjectSet<Publication> publications = db.query(
    new Predicate<Publication>() {
        public boolean match(Publication publication) {
            return publication.getYear() > 1995;
        }
    }
);
for (Publication publication: publications) {
    System.out.println(publication.getTitle());
}

In the Simple Object Data Access (SODA), the database is seen as a tree. To get an object, it is needed to explore the tree. A constraint can be set on each node. Different methods are provided and are self-explanatory:

- descend
- constrain
- sortBy
- orderAscending
- orderDescending
- execute

For the constraints:

- greater and smaller
- identity, equal and like
- and, or and not
- startsWith and endsWith
- contains
ObjectContainer db = Db4oEmbedded.openFile("test.db");
// find all publications after 1995
Query query = db.query();
query.constrain(Publication.class);
query.descend("year").constrain(Integer.valueOf(1995)).greater();
ObjectSet<Publication> publications = query.execute();
for (Publication publication : publications) {
    System.out.println(publication.getTitle());
}

// find all publications of author "Moira C. Norrie"
Query query = db.query();
query.constrain(Publication.class);
Author proto = new Author("Moira C. Norrie");
query.descend("authors").constrain(proto).contains();
ObjectSet<Publication> publications = query.execute();
for (Publication publication : publications) {
    System.out.println(publication.getTitle());
}

To update a persistent object, it is simple. First, the object is retrieved through
one of the three ways described above. A method is called on the object to modify
it just like it is normally done. Finally, the modified object can be written back in
the database through the call to the store method.

ObjectContainer db = Db4oEmbedded.openFile("test.db");
// retrieve existing object
Author michael =
    db.queryByExample(new Author("Michael Grossniklaus")).next();
// update object in memory
Calendar calendar = Calendar.getInstance();
calendar.set(1976, Calendar.JUNE, 22);
michael.setBirthday(calendar.getTime());
// update persistent object
db.store(michael);

In the background, db4o needs to manage the links between objects in the mem-
ory and the same objects in the database. It saves IDs which are weak references.
When an object is retrieved, created or saved, it becomes a fresh reference. Like
that, db4o can test those references, see if an object has been modified or not and
if this the case, the object in the database is updated.

Deleting an object is similar to the update. The reference must be fresh. After
retrieving of the object, the method to call is simply delete of ObjectContainer
which removes the object.

ObjectContainer db = Db4oEmbedded.openFile("test.db");
// retrieving author "Moira C. Norrie"
Author moira = db.queryByExample(new Author("Moira C. Norrie")).next();
// deleting author "Moira C. Norrie"
db.delete(moira);

The objects referenced in an object are treated differently depending on the action:

**Store new objects** All the referenced objects are saved to the database.

**Update objects** Only primitive and string values are updated.

**Delete objects** Referenced objects must be deleted manually.

In fact, for the updating procedure and the deleting procedure, the configuration can be changed to take these objects into account: `cascadeOnUpdate` and `cascadeOnDelete` from `ObjectClass`, so that a part of the object graph is traversed. In the case of the update, it is possible to define the depth of the update. When deleting objects, it is sometimes useful to resync the data between the memory and the database. Indeed, the `delete` method only deletes objects in the database. The method `refresh` of `ExtObjectContainer` is provided to avoid inconsistencies.

### 3.1.2 OO Programming in Database

db4o manages the different structures present in the OO programming:

- hierarchies
- inheritance and interfaces
- multi-valued attributes, arrays and collections

It has also its own collections created specially for this database which implements the transparent persistence and the transparent activation: `ArrayList4` and `ArrayMap4`.

Transparent persistence allows the database to update stored objects automatically when they are modified. The `store` method must be called only once. Therefore, the application layer does not need to care about the object persistence anymore.

To enable transparent persistence:

```java
config.add(new TransparentPersistenceSupport());
```

When an object is retrieved, not all its object fields are loaded in the memory. This done with a certain depth which is configurable per class: objects can reference to other objects which reference to other objects and so on. This is the activation. All the other fields are set to `null`.

There is an option to make activation transparent to the application. The different fields will be automatically activated when needed.

```java
config.add(new TransparentActivationSupport());
```
3.1.3 Transactions

The transactions in db4o respect the ACID transaction model. Moreover, data transactions are journalized to avoid loss of data. The db4o core works in single-thread mode so there is no problem with multiple interactions happening simultaneously. As said before, an ObjectContainer is transactional. Opening it automatically start a transaction.

To write data explicitly in the database, the commit method of ObjectContainer is called. It is important to note that closing the ObjectContainer implicitly commits the modifications.

```java
ObjectContainer db = Db4oEmbedded.openFile("test.db");
// retrieving author "Moira C. Norrie"
Author moira = db.queryByExample(new Author("Moira C. Norrie")).next();
// creating author "Stefania Leone"
Author stefania = new Author("Stefania Leone");
// creating new publication
Publication article = new Publication("Web 2.0 Survey");
article.addAuthor(stefania);
article.addAuthor(moira);
// storing publication
db.store(article);
// committing
db.commit();
```

On the contrary, to avoid data changes in a transaction to be written to disk, the rollback method of ObjectContainer is used.

```java
ObjectContainer db = Db4oEmbedded.openFile("test.db");
// retrieving publication
Publication article =
db.queryByExample(new Publication("Web 2.0 Survey")).next();
// updating publication
Author michael = new Author("Michael Grossniklaus");
article.addAuthor(michael);
db.store(article);
// aborting transaction
db.rollback();
```

The objects in memory are not changed back to their initial state so that inconsistencies could appear between the disk and the memory. The resync needs to be done manually by calling:

```java
db.ext().refresh(article, Integer.MAX_VALUE);
```
3.1.4 Collision

db4o uses the read-committed isolation level: data changes made in other transactions can only be read after their commit. To avoid collisions and inconsistencies, checking if an object has changed during a transaction can be helpful. The method peekPersistent of ExtObjectContainer allows to get the version of an object stored in the database.

```java
ObjectContainer db =
Db4oClientServer.openClient("localhost", 3927, "...", "...");
// retrieving author "Moira Norrie"
Author moira = db.queryByExample(new Author("Moira C. Norrie")).next();
// storing initial value of field
Date birthday = moira.getBirthday();
...
// retrieve stored value of field
Author persisted = db.ext().peekPersistent(moira, 9, true);
// compare the values and abort if necessary
if (persisted.getBirthday() != birthday) {
    db.rollback();
} else {
    db.commit();
}
```

The other way is to use semaphores to lock objects. The semaphore is defined with an unique name and by the time to wait if it is already locked by another transaction.

```java
ObjectContainer db =
Db4oClientServer.openClient("localhost", 3927, "...", "...");
if (db.ext().setSemaphore("SEMAPHORE#1", 1000)) {
    // critical code section
...
    // release semaphore after critical section
    db.ext().releaseSemaphore("SEMAPHORE#1");
}
```

3.1.5 The Ways the Database Operates

In db4o, two modes are available:

- **Embedded mode**: The application and the database are on the same machine. In this mode, a single thread or a multi-thread system can be chosen. Firstly, the direct file access (fig. 3.1) allows only one user and one thread to interact with the database which is accessed directly. The embedded object container is used for each operation. Secondly, using client session (fig. 3.2), the database operations are this time executed in session object container. Therefore, it is multithread.
Client/Server Mode: The database is located on a server which can be accessed by clients to do some operations on the database.

In the networking mode (fig. 3.3), the client simply connects to the server using TCP/IP. All the operations are limited to methods of ObjectContainer: query, insert, update and delete. The ObjectContainer communicates with the ObjectServer that operates directly on the database. To be able to send commands, the Out-of-Band Signalling (fig. 3.4) must be used. The client sends a message through the ObjectContainer and the ObjectServer processes the message.
3.1.6 Replication

Replication is a mechanism to have different copies of a database on different servers to ensure availability and the reliability of the data. The master server contains the original database which will be replicated to the client servers. Therefore, it is important to keep all those databases synchronized.

The db4o database does not provide a way to set up an automatic replication in the configuration. This needs to be programmed in the application. But an interface to different replication modes is available.

Indeed, this database supports three of them.

- Snapshot Replication: The images of the master database are replicated to clients depending on a state or on a schedule.

- Transactional Replication: Everytime there is a modification in the master database, the clients are updated.

- Merge Replication: Modifications in a client are sent back to the master server and the other clients are then updated.

The replication system requires three steps:

1. IDs and version numbers are generated.

2. a ReplicationSession object is created.

3. the objects are replicated.

The direction of the replication can be chosen with the setDirection method. The replicate method is used to put the new object in the database with the out-of-date object.

```java
// configuration
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.file().generateUUIDs(ConfigScope.GLOBALLY);
config.file().generateVersionNumbers(ConfigScope.GLOBALLY);
ObjectContainer db1 = Db4oEmbedded.openFile(config, "test1.db");
```
ObjectContainer db2 = Db4oEmbedded.openFile(config, "test2.db");
// replication session
ReplicationSession replication = Replication.begin(db1, db2,
        new ReplicationEventListener()
        {
            public onReplicate(ReplicationEvent e) {
                if (e.isConflict()) {
                    e.overrideWith(e.stateInProviderA());
                }
            }
        });
replication.setDirection(db1, db2);

// update database and replicate (transactional replication)
Author stanzetta = new Author("Christoph Zimmerli");
db1.store(stanzetta);
replication.replicate(stanzetta);
replication.commit();
// replicate changed publications (snapshot replication)
ReplicationProvider provider1 = replication.providerA();
ObjectSet<Publication> result =
        provider1.objectsChangedSinceLastReplication(Publication.class);
for (Publication publication: result) {
    replication.replicate(publication);
}
replication.commit();
// close the replication session
replication.close();

3.1.7 Schema Evolution

The structure of the database can be modified to reflect a change in reality for
example. In fact, the class definitions and hierarchy are related to the database
schema. In programming language, there is a notion called Refactoring which can
also be used in object databases. This can automatize changes in the database
structure. There are different kinds of refactoring:

- Changes to interface.
- Removing a field.
- Adding a field.
- Changing the type of a field.
- Renaming a field.
- Renaming a class.
• Merging fields.
• Splitting fields.
• Moving fields.

### 3.1.8 Callbacks

The database can react to different events. It is possible to call methods before (prefix `Can`) and after (prefix `On`) an event. All these methods are in the `ObjectCallbacks` interface.

```java
package com.db4o.ext;
public interface ObjectCallbacks {
    public boolean objectCanActivate(ObjectContainer c);
    public boolean objectCanDeactivate(ObjectContainer c);
    public boolean objectCanDelete(ObjectContainer c);
    public boolean objectCanNew(ObjectContainer c);
    public boolean objectCanUpdate(ObjectContainer c);
    public void objectOnActivate(ObjectContainer c);
    public void objectOnDeactivate(ObjectContainer c);
    public void objectOnDelete(ObjectContainer c);
    public void objectOnNew(ObjectContainer c);
    public void objectOnUpdate(ObjectContainer c);
}
```

Just like the triggers in relational databases, there are some scenarios possible: recording or preventing updates, setting default values, object integrity, …
3.2 Versant Object Database

3.2.1 Architecture

The architecture is described in the figure 3.5. There are several interfaces (C, C++, Java, ...) to communicate with the Object Manager. It is able to interact with the Versant Server through the Network Layer. The Versant Server takes care of the data on the different devices. The architecture is dual cache and multi-threaded which is optimized for currently used servers such as a multi-core server.

In the figure 3.6, the two caches can be seen. The first one is the Object Cache in the client whereas the second one, the Page Cache is on the server. The Page Cache works like a classical cache. In the beginning, it is empty. After some accesses to the databases, it can be full so some pages need to be erased. Before that, it must be verified that these pages are already up-to-date in the database. As any cache, the data are modified in the cache before being saved in the database. The Versant Server first checks if the data needed are in the Page Cache. If it is not the case, it will load the pages containing the data in the cache.

The figure 3.7 shows how the multi-thread works. The server has a thread for each Session Object Cache which is linked to one or more client threads. Each client can have several threads wishing to access the Versant database.
Figure 3.6: Versant dual cache architecture

Source: [13]

Figure 3.7: Versant multi-threaded architecture

Source: [13]
3.2.2 Java Versant Interface

The Java Versant Interface (JVI) is, as the name implies, the interface for the Versant database in the Java programming language. Previously, different interfaces have been mentioned. In this writing, the focus will be on the JVI.

![Versant JVI architecture](source: [13])

JVI offers a way to store and access persistent Java objects and thus, working in the Java Virtual Machine (see fig. 3.8). The syntax and semantics are in Java. It works with the Java garbage collector and multiple threads. The architecture is client-server based. The access to the Versant object database is obviously provided. To increase the performance, objects are cached in the client whereas the queries are done on the server.

JVI is composed of three layers:

- Fundamental Layer
- Transparent Layer
- ODMG Layer

**Fundamental Layer**

It gives a direct access to Versant ODBMS functionality. The sessions, attributes and queries are seen as Java objects but this is not the case for the persistent objects in the database. The method `get()` and `put()` are used to access their attributes. In fact, this layer is database-centric and is located in the package `com.versant.fund`. 

---

21
Transparent Layer

The **Versant** object model is blended into the **Java** language. The programs can be written the normal way and the **Java** objects can be persistent and transactional. To make an object persistent, it should simply be indicated in the declaration of the class and the enhancer should be used. So, the persistent objects can be treated as any other **Java** objects. The interactions with the database are done transparently and in accordance with the database: the mapping between persistent **Java** objects and the **Versant** object model is done automatically. This makes this layer language-centric and it is on top of fundamental binding. It is defined in the package `com.versant.trans`.

ODMG Layer

This layer simply enables the use of the standard interface methods and data constructs specified in the Object Database Management Group 2.0 and 3.0 Java binding specifications.

### 3.2.3 Development steps

1. Write **Java** classes with persistence and database in mind.
2. Write configuration for enhancer program.
3. Compile **Java** classes.
4. Run enhancer to apply the persistence defined in the configuration file. Code is generated.
5. Create the database.
6. Launch the program.

### 3.2.4 Persistence

The persistence in **Versant/Java** is by reachability, so there must be a way to navigate through the graph of an object. To make it easier, it is possible to choose an object as a database root and give it a name to retrieve it easily. This functionality can be used in the transparent and ODMG layer. There are three methods to remember:

- `makeRoot()` The object becomes a root and is stored.
- `deleteRoot()` The object loses its place as root.
- `findRoot()` A root object is retrieved.

In fact, the navigation is transparent and is not manually defined by the user. It can start from an identity, a root, a class extent or a query. To go from an object to another can only be done through links between objects. **Versant** takes care of the locking and retrieving of objects. It can work with several databases and is not limited by a database boundaries.
First and Second Class Objects

In Versant, each persistent object can be defined as First Class Objects or Second Class Objects. Then, the object persistence is treated differently depending on which category it belongs to.

- **First Class Objects (FCO)**:
  They can be stored and retrieved individually. They have an ID which is called Logical Object Identifiers (LOID). Besides, the changes made to the objects are automatically saved and the reference to existing FCO always works. Transient fields are obviously not saved in the database.
  
  There are two categories of persistence for FCO:

  **Persistent always (p)** The object is persistent when it is instantiated. When it is modified, it is marked as dirty.

  **Persistent capable (c)** At the instantiation, the object is not stored in the database but it can be in the future by using `makeRoot()`, `makePersistent()` or because of the persistence by reachability. When it is modified, it is marked as dirty.

  That leads to this rule: whenever a class is persistent always or persistent capable, its parent must also be persistent always or persistent capable. This is not true for an `Object` class parent.

- **Second Class Objects (SCO)**:
  SCO can only be stored as a component of a FCO which is its owner. Each SCO can only have one owner, it cannot be shared. If an attribute of the SCO does not have an attribute type defined in Versant, then it is saved as serialized Java byte stream. So queries cannot retrieve them directly.
  
  For SCO, there are three categories:

  **Transparent dirty owner (d)** When an object is modified, the owner is marked as dirty. It is useful for serialized collections.

  **Persistence aware (a)** The attributes of an FCO can be changed directly. To save the changes made to an SCO of a FCO, the method `dirtyObject()` must be called on its FCO owner.

  **Not persistent (n)** The object is not persistent, so there is no code enhancement and it is not possible to access the fields of a persistent object.

### 3.2.5 Transaction

The operations on database are performed in sessions. As long as the session is not closed, Versant is in a transaction. A new transaction is created after the `commit()` and `rollback()`. The method `endSession` closes the current session and makes a commit of the last transaction.

There is a session implementation in each JVI layer. On the client side, an object
cache and a cached object descriptor table are present. On the server side, there is a page cache containing recently accessed pages.

The properties of a transaction are:

- ACID:
- coordination: the objects are locked in case of multiple users accessing it.
- distribution: multiple databases.
- presence: application code is always contained in a transaction.

The commit() method which releases locks and stores the contents of the cache is not the only method to apply modifications to the database. There are also the checkpointCommit() method which retains locks and retains cached objects and the commitAndRetain() method which releases locks and retains cached objects.

Here is an example for the creation of a persistent object.

```java
// use the transparent layer
TransSession session = new TransSession("PublicationsDB");
// find a previously defined root
Set< ? > publications = (Set< ? >) session.findRoot("publications");
// create a new author assuming that the Author class
// is either "p" or "c"
Author moira = newAuthor("Moira C. Norrie");
for(Object object: publications) {
    Publication publication = (Publication) object;
    publication.addAuthor(moira);
}
// commit the changes
session.commit();
// end the session
session.endSession();
```

When a persistent object is created, it is created as a Java object and its database information are put in the Versant object cache. In the commit, the object is stored in the database but if the rollback is called, it is destroyed. The query for an object returns a set containing a proxy object for every object corresponding to the query. Then, the access to an object in the database is done transparently; Versant retrieves the object or de-serializes it.

### 3.2.6 Object Modifications in Database

As seen previously, a distinction must be made between First Class Objects and Second Class Objects.

The first case is when the object is updated.

- For FCO, everything is automatic and transparent. Indeed, the changes are stored in the database after commit.
• For SCO, if the object is in the category of Transparent Dirty Owner, it works exactly as if it is a FCO. In the case of a Persistent Aware SCO, the dirtyObject() must be used on the FCO owner.

The second case is when the object is deleted.

• For FCO, it must be deleted explicitly using TransSession.deleteObject() or TransSession.groupeDeleteObjects() methods. The deletion are applied to the database objects whereas the garbage collector takes care of the Java instances.

• For SCO, to delete an object, its reference must simply be set to null and the garbage collector will erase it. At the commit, the FCO will be saved without the deleted SCO.

### 3.2.7 Supported Collections

Different types of collections can be used to contain persistent objects in Versant:

- The classical Java collections: Array, Vector, Hashtable, LinkedList.
- FCO collections: VVector, VHashtable.
- SCO collections: DVector.
- Scalable large collections: LargeVector.
- ODMG collections.

FCO collections, SCO collections and ODMG collections are entirely loaded, which can result in a performance issue when the collections are really big. On the other hand, LargeVector which implements the interface of Vector, is splitted into several nodes. So, only needed nodes will be loaded when a element of the collection is accessed. Some nodes are then locked and not the whole collection. Multiple users can have access to the collection but this situation is sensitive to deadlocks. That is why a locking protocol must be defined.

ODMG collections are implemented following the specifications of the ODMG standard. As seen before, they are in the ODMG layer in Versant. ODMG 2.0 and ODMG 3.0 are supported. These collections are FCO. But, above all, they provide some additional queries capabilities such as:

- boolean existsElement(String predicate)
- DCollection query(String predicate)
- Iterator select(String predicate)
- Object selectElement(String predicate)

The predicate corresponds with the where from a VQL query (see section 3.2.8). These queries only works with persistent collections.
3.2.8 Versant Query Language

To make queries to the database, Versant has developed Versant Query Language (VQL). The VQL 6 is simply a subset of OQL ODMG 2.0 and is very limited. The latest version, VQL 7, is more expressive, supports server-side sorting and has better indexing.

Queries are written as strings. They are compiled, optimised and then executed on the server. It is possible to use parameters in the queries just like prepared queries in SQL: they start with a dollar symbol ($) and to attribute values to them, the bind() method is used.

Here is an example using VQL queries:

```java
// create a new publication, assuming the Publication class is "p"
Publication pub = newPublication("Web 2.0 Survey");
// find authors Stefania Leone and Moira C. Norrie
String queryString = "select selfoid from Author where name = $name";
Query query = newQuery(session, queryString);
query.bind("name", "Stefania Leone");
QueryResult result = query.execute();
Object author = result.next();
if(author != null) {
    pub.addAuthor((Author) author);
}
query.bind("name", "Moira C. Norrie");
result = query.execute();
author = result.next();
if(author != null) {
    pub.addAuthor((Author) author);
}
```

3.2.9 Events

To notify clients of some changes in the database, Versant has integrated different event types and listeners. All listeners for each event type are defined as sub-interfaces of VersantEventListener. The different types of event are:

- **Class Events**: when an instance of a specified class is created, modified or deleted.

- **Object Events**: when an object or a group of objects are modified or deleted.

- **Transaction demarcation**: when a transaction begins or ends.

- **User-defined Events**

Events are propagated through event channels. So, event listeners must be connected to a channel and the application must do the same to receive the event. Global namespaces are used to avoid differences for each client connected to the database. There are also different types of event channel:
• Class-based: used to notify a set of classes of class events.
• Object-based: used to notify a set of objects of objects events.
• Query-based: used to notify objects from a query of class events.

On the client side, the class `EventClient` manages event channel and receives the events. It can define a new channel or access an existing one thanks to its name.

It is also possible to define hooks on persistent objects. These events can happen at any state of an object:
• `activate()` and `deactivate()`.
• `preRead(boolean act)` and `postRead(boolean act)`.
• `preWrite(boolean deact)` and `postWrite(boolean deact)`.
• `vDelete()`.

The parameters `act` and `deact` allows to act differently whether the object has been (de)activated or not. They can be used to attribute values to transient attributes or to preserve referential integrity just like the following example.

```java
public class Author {
    private String name;
    private Date birthday;
    private Set<Publication> authoredBy;
    // delete hook that also removes publication of the deleted author
    void vDelete() {
        TransSession session = (TransSession)
            TransSession.sessionOfCurrentThread();
        for (Publication publication: this.authoredBy) {
            session.deleteObject(publication);
        }
    }
    ...
}
```

### 3.2.10 Schema Evolution

Schema evolution depends on the programming interface. Indeed, in the fundamental layer, it must be done manually. Different actions are possible: inserting, appending, dropping and renaming attributes and adding and renaming classes. In the transparent layer, the `setSchemaOption(int)` allows the automatic schema evolution. It is convenient to avoid incompatibilities between a schema object and the corresponding Java class. There are three options available:

• **SCHEMA_ADD_DROP_ATTRIBUTES**: The database class is modified to have the same attributes as the Java class.
• **SCHEMA_FORCE_DROP_DATABASE**: The database class is dropped, all the instances of the class are deleted and a new class is created to match the Java class.

• **SCHEMA_THROW_EXCEPTION_ALWAYS**: If there is an incompatibility, an exception is raised.
3.3 Progress ObjectStore

3.3.1 Virtual Memory Mapping

The architecture ObjectStore is page-based. It works as an extension of the operating system virtual memory. It is called virtual memory mapping architecture. The key points are:

The logical address and the physical address The reference to an element in the database is divided into 4 parts: which database, which segment, which cluster and the offset in the cluster. This leads to a space of $2^{128}$ addresses. So, the data needs to be mapped from this space to a portion of the database client application’s virtual memory which is called the persistent storage region (PSR). It is really small compared to that space.

The physical memory and the secondary storage If data are needed by the client, these data must be located in PSR. Therefore, data must first be copied to the PSR before being able to use it. On the other hand, recently accessed data are stored in the cache which serves as a secondary storage.

The page faulting As stated above, memory is small and all the pages cannot be transferred to the client. When a page fault happens, ObjectStore checks the cache contents. If it finds the data needed, they are simply paged into memory from cache. If not, it must retrieve data from database and puts them in the cache and then in the application.

The address translation It is the conversion of address from logical space to physical space and inversely. It can happen at different stages. First, it is done when data are retrieved from the database to the cache. Secondly, if PSR is nearly full, a retranslation is needed. Finally, updated data (and more precisely pages containing these data) must be translated back to its logical address and then stored back in the database.

3.3.2 Architecture

The properties of the ObjectStore architecture are:

- It is virtual.
- It is shared.
- It is distributed.
- It is heretogeneous.
- It is persistent.
- It is transactional.
The schema of the client-server architecture can be seen on the figure 3.9. The C++ case will be considered in this writing. On the server, there are three different elements:

**ObjectStore Server** Its role is to transmit pages and manage the ACID semantics using page permits (see section 3.3.4). If there are other servers, it must manage the sharing of pages. Restarting a server automatically launches a recovery mechanism.

**Database** It is managed by only one server. It is composed of binary files used to store pages of data which are in fact C++ objects. These ones are stored in the same format as they are used in the memory. Moreover, they are accessed and updated with the same C++ syntax as heap allocated objects. The database is generally placed in the server hard drives.

**Transaction Log** It belongs to only one server, so, if there are several of them, there must be several transaction logs. All the updated pages are written to the transaction log and these pages are only written back to the database whenever a transaction commits. This log is used for automatic recovery and faster commits.

On the client, the components are:

**C++ Client** It is a C++ application which is linked with the ObjectStore libraries. It interacts with the database and manipulates retrieved objects. Besides, the pages are automatically transferred from the database when needed.

**Cache** One cache memory mapped file is associated with one client process. Its size is fixed upon startup and cannot be changed without stopping the client.
It contains all pages from the database once needed by this client. They can even stay in the cache between transactions.

**Commseg** One commseg memory mapped file is associated with one client process. It only contains meta-information about the pages in the cache. The information on the **permit** and the **lock** of every page in the cache is stored in this file. The permits can also be kept between transactions.

**Cache Manager** One cache manager is present on the client machine. So, all the clients on the same machine are connected to it. Its function is to deal with the permit revocation and with the cache and commseg files. However, the page claiming is not its responsibility.

**Persistent Storage Region** It is a region of the virtual address space in the **C++** application. All the persistent objects needed by the client have their address mapped into this space. Therefore, if the value of a pointer is inside the **PSR**, that simply means that it points to a persistent object. The **PSR** is cleared at the end of every transaction and so, is ready for the next transaction.

### 3.3.3 Page Fetching

This section will focus on the way **ObjectStore** manages the fetching and the mapping of pages to be used by the client. As mentioned above, when the client needs pages, they are automatically retrieved and put in the client cache. The logical addresses of the fetched pages are transformed into physical addresses inside the **PSR**: the reference in the database is transformed into a **C++** pointer which points within the **PSR**. It does not need to point to already present objects, it can be used to make a reservation of the space for future objects.

To keep the transaction consistent, server permits and clients locks (see section 3.3.4) are delivered automatically. When there is not enough space in cache, existing pages are removed to leave space for new pages. The pages which have been modified are sent to the server to save the modifications in the database. But if they are read-only, they are simply erased. Indeed, a copy already exists in the database.

The figure 3.10 illustrates the steps to get a page X.

1. **ObjectStore** installs a **SIGSEGV** handler.
2. The application gets a pointer **p** to an object which is located on page X.
3. **p** is dereferenced so the **SIGSEGV** handler is called.
4. The page needed is located by looking into the virtual mapping table and then, is retrieved from server and stored in the cache.
5. Finally, the page X is mapped to the **PSR** and the execution of the program can continue.

---

1 It is a signal sent to process when it tries to access an invalid memory address or when there is a segmentation fault.
3.3.4 Page Permits and Locks

To keep high performance in ObjectStore, different choices in the architecture have been made. Firstly, data is cached between transactions. Secondly, the number of times that a lock is acquired is reduced. Finally, the data in the cache remains globally consistent.

In fact, in ObjectStore, there are two types of locks:

- **The transaction locks** It is the state of a page during a transaction.

- **The ownership permits** It is the state of a page which is located in the cache.

The server is the one that maintains the permits and transmits them when a client needs a page: a permit is associated with a page. The client can then request the lock of the page if necessary.

The table 3.1 shows the different possibilities of permit and lock on a page and then the response from the cache manager (positive or negative). When a client receives a read permit on a page, it can lock it for reading without asking the server. So, it does not limit the number of clients accessing the page for reading. This is different for the write permit, only one client can write to a page at a given moment: the permits of the other clients must be revoked. When receiving this permit, it can lock a page for reading (the permit becomes a read permit) or writing.
Table 3.1: The combination of permits and locks.

<table>
<thead>
<tr>
<th>Permit</th>
<th>Lock</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>✗ server only calls back permit if other client needs to write</td>
</tr>
<tr>
<td>read</td>
<td>no lock</td>
<td>✓</td>
</tr>
<tr>
<td>write</td>
<td>read</td>
<td>✓ permit for page downgraded to read</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>✗</td>
</tr>
<tr>
<td>write</td>
<td>no lock</td>
<td>✓</td>
</tr>
</tbody>
</table>

The server manages a table of permits given to the clients. When a client requests a page from the database, the server must check the page permits assigned the other clients. If one of them has permits in conflict with this one, the server must then take back the permits for this page of the other clients. The cache manager (which is in the client) checks the permit and lock of pages for callback. It responds with **POSITIVE** if the lock is less restrictive than the permit. Otherwise, it responds with **NEGATIVE** and a flag is raised so that the revocation of the permit must occur after the transaction.

Figure 3.11: A classical scenario of permit revocation.

The whole mechanism of caching data and associated permits is what makes it really efficient to keep consistency between cached data and the same data in the database. It can be seen as a shared virtual memory.
3.3.5 Distribution and Heterogeneity

The clients can retrieve objects located in different remote databases and this can be done in the same transaction.

![Diagram of different servers and clients running on different platforms.](source)

Figure 3.12: A example of configuration with different servers and clients running on different platforms.

Source: [23]

The platforms of the clients and the servers do not matter. The objects will be transformed by the client when the page is mapped to the cache. Also, the database keeps records about the last platform writing to each page.

3.3.6 Persistence

ObjectStore provides persistence by instantiation (In Java, the persistence is by reachability). Working with persistent objects is as simple as working with in-memory objects. Indeed, ObjectStore can be considered as a library offering services for accessing the database. To create a persistent object, the operator `new` is called and an address within `PSR` is returned, so the persistence is defined at the creation of the object. In fact, this is an overloaded `new` operator which takes three arguments:

- The location of the object.
- The type of the object.
- The number of objects

2This argument is optional and is only used for arrays.
It is possible to allocate object transiently on the heap, in a database, in a segment, in a cluster or next to another object. The advantage of being able to allocate an object persistently or transiently is that the same class definition can be used in the two cases. This is independent to the type of the object: the persistence in ObjectStore is said to be orthogonal to the type of the object. The possibility to choose where to put an object in the database can give higher performance. For example, objects used together can be clustered so that they are near each other.

### 3.3.7 Database Layout

In fact, the database (see fig. 3.13) is composed of segments which are themselves composed of clusters. These clusters contain a certain number of pages.

![Figure 3.13: The database and the segments.](source: [13])

The segments are a way to define a logical partitioning of objects whereas the clusters are used to regroup pages containing objects which are often used together. There are several special segments. The segment 0 contains the database schema and the database roots. The segment 2 is the default segment and the segment 4 is the first segment that can be created by the user. Moreover, there is a maximum of $2^{32}$ segments in one database.
Every segment has a default cluster which is the number 0 and the others are created by the user. A segment contains a maximum of $2^{31}$ clusters.

### 3.3.8 ObjectStore Programming

Different ObjectStore libraries are available to the developer.

- `objectstore`
- `os_database`
- `os_transaction`
- `os_typespec`
- `os_database_root`
- `os_segment`
- `os_cluster`

To use persistent classes in a program, these classes must of course be written but a schema file is also needed. It indicates which classes should be made persistent. It is called `schema.osg` by convention. Here is an example of it in which the class `Foo` is persistent.

```plaintext
#include <ostore\manschem.hh>
#include <ostore\ostore.hh>
#include "Foo.h"
OS_MARK_SCHEMA_TYPE(Foo)
```

This file is then compiled using `ossg` compiler. Finally, the program can be compiled using the C++ compiler.

Several methods are provided by `os_database` to manage the database:

- `create()`: To create a new database.
- `open()`: To open an existing database.
- `save()`: To save the changes to the database.
- `close()`: To close a database and invalidate pages in the cache.
- `destroy()`: To delete a database.

```plaintext
#include <os_pse/ostore.hh>
int main(int argc, char **argv, char **envp)
{
    objectstore::initialize();
    os_database *db = os_database::create("publications.db", 0664, 1);
    ...
    db->save();
} 
```
db->close();
db->destroy();
objectstore::shutdown();
}

To navigate through database, a departure point (which is an object) is needed. The result of a query can be used but also the database roots. Indeed, some objects in the database can be given a name which can be used to access easily these objects.

```cpp
Author* Database::retrieveAuthor(const char *name)
{
    os_Dictionary<char*, Author*> *authors = 0;
    os_database_root *root = _db->find_root("authors");
    if (!root) {
        root = _db->create_root("authors");
        authors = new(_db, os_Dictionary<char*,
                      Author*>::get_os_typespec())
                       os_Dictionary<char*, Author*>();
        root->set_value(authors);
    }
    authors = (os_Dictionary<char*, Author*>*) root->get_value();
    return authors->pick((char*) name);
}
```

The database roots are of class `os_database_root`. They are composed of two parts: the root name and a pointer to the object.

### 3.3.9 Transaction

Just like other databases, persistent objects are modified or accessed within a transaction. There is also a commit and a rollback mechanism. ObjectStore respects the ACID properties of transactions.

- **Atomicity**: The commit guarantees that all data modified in a transaction will be written and will be recoverable whereas the rollback guarantees that all the changes will be undone.
- **Consistency**: Updates cannot be applied or lost as long as pages is being written to the database.
- **Isolation**: ObjectStore uses a two-phase locking protocol. It gets locks when needed, keeps them during a transaction and releases all of them at the end.
- **Durability**: This is guaranteed because ObjectStore first writes the modified pages in the transaction log. Then the database is updated in the background. If there are problems, thanks to this transaction log, all the committed pages can automatically be recovered.
Moreover, transactions can be of type read or write. This is linked to the page permits and locks. An exception happens when a read transaction tries to access to lock a page for writing. They can be local or global. In the first case, only the thread beginning the transaction can execute. In the second case, a transaction can be shared by all threads in a session. The transactions are lexical or dynamic. The lexical ones are automatically relaunched when there is deadlock and their beginning and their end must be in the same code block. They are also thread-local. For multithread applications, the dynamic transactions are preferred.

```cpp
objectstore::initialize();
os_transaction::initialize();
os_database *db = os_database::open("publications.db", 0, 1);
OS_BEGIN_TXN(txn0, 0, os_transaction::update)
{
    ...
    os_transaction *txn = os_transaction::get_current();
    txn->abort();
}
OS_END_TXN(txn0)
```

A transaction is surrounded by `OS_BEGIN_TXN` and `OS_END_TXN`. The type of the transaction is indicated at the beginning of the transaction. Nesting of transactions is possible. Manipulating transactions can be done using the `os_transaction` class. They are then dynamic. If `ObjectStore` macros are used, they are lexical.

### 3.3.10 Manipulating Persistent Objects

As stated before, an overloaded `new` operator is used to create persistent objects. This example shows how to create and delete an object of class `Foo`.

```cpp
os_database* db = os_database::open("d:/x.odb");
OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    Foo* f = new (db, ts<Foo>()) Foo();
    //...other code...
    delete f;
}
OS_END_TXN(use_case1)
db->close();
```

To retrieve the type of the object, the `ts<>` method is called. The element returned is a `os_typespec` object. It is used to determine a persistent type in `ObjectStore`. Modifying or deleting a persistent object is done by using standard C++. The changes will be written to the database when the pages are sent back to the server. It is thus completely transparent to the developer.
3.3.11 Collections

ObjectStore provides their own types of collections which can be templated or not (see fig. 3.14). They correspond to the os_collection class which offers all the functionalities of the classical collections.

![ObjectStore Collections Diagram]

Figure 3.14: The ObjectStore collections.

Source: [13]

Here are examples on how to:

- create a collection.

```cpp
Author::Author(const char *name)
{
    ...
    _authors = new(os_cluster::of(this),
                    os_Set<Publication*>::get_os_typespec())
                os_Set<Publication*>();
}
```

- manipulate a collection.

```cpp
void Author::addPublication(const Publication *p)
{
    _authors->insert((Publication*) p);
    os_List<Author*> *authoredBy = p->_authoredBy;
    authoredBy->insert(this);
}
```
• delete a collection.

Author::~Author(void)
{
    ...
    delete _authors;
    _authors = 0;
}

Cursors are an easy way to navigate through collections. They are represented by the os_Cursor class. There are three methods:

first() It changes the position of the cursor to the first element.

next() The target of the cursor is changed to the following element.

more() It returns true if the position of the cursor corresponds with an element.

// Find all authors younger than 30 with more than 10 publications
char* query = "this->getAge() < 30 && this->getPublicationCount() > 10";
os_Set<Author*> &result = _authors->query("*Author", query, _db);
os_Cursor<Author*> c(result);
for (Author *author = c.first(); c.more(); author = c.next()) {
    cout << author->getName() << endl;
}

Collections offer the possibility to query them. The query() method requires as arguments, the object type, the query string and the database. The query string can be defined in C++ or as a regular expression. But the function calls in it are only limited to basic types. Nested queries are supported.

3.3.12 Schema Evolution

In ObjectStore, there are two kinds of schema. The program schema contains all the definitions of classes in a program whereas the database schema contains all of the ones in the database. It is the program schema that will add its own defined types inside the database schema and thus new types will be accepted by the database.

To prevent data corruption in the database, everytime a program tries to access a database, their two schemas are compared to see if there are incompatibilities. If this is the case, an exception is raised.

There is an utility called ossevol which can migrate an existing database schema to a new program schema. The mapping is automatically generated and is applied on the objects of the database which need to be changed to still be compatible. The possibility to configure a personalized evolution is offered by the ossevol API.

However, schema evolution is not required when new classes are added to database. Indeed, these new classes are transparently added to the database schema at the database access or at the creation of instances of these new types in the database.
3.4 Objectivity\[25, 27\]

3.4.1 Architecture

The architecture of Objectivity (see fig. 3.15) is client/server and distributed. The work and data can be distributed transparently across the network. Moreover, there are different specialized server processes:

**Data Server** It transfers data from disk to the application.

**Query Server** It improves performance when using queries.

**Lock Server** It manages permissions and locks when accessing data.

![Objectivity Architecture Diagram](image)

Figure 3.15: Objectivity architecture.

This architecture provides scalability and high availability. A database can be divided into autonomous partitions and some data and even entire databases can be replicated on different places in the network.

Objectivity provides Parallel Query Engine (PQE). It gives the ability to perform queries quickly on a big amount of data. It takes advantage of the distributed architecture of Objectivity. Indeed, the search can be done with different query servers working on different parts of the data (which can be different databases). It is very customizable. The task splitter can be configured to target some databases and some containers and an objects filtering can be done at the end of the query. Finally, other databases and search engine are accessible by going through the gateway.

The performance is further improved by clustering data, using multi-dimensional index and client data caching. Objectivity also provides scalable collections which are independent of the programming language.
3.4.2 Storage Model

In **Objectivity**, the databases are grouped into a **federated database** (see fig. 3.16). It is the highest level in the logical storage model. It contains the data model and a list of the databases which are in it. In fact, federated database and databases are represented physically as files. A federation can hold up to 65530 databases whereas a database can hold up to 65530 containers. A database has a default container, the others are created by the user. As stated above, **Objectivity** is distributed, so a database can easily be "split" for parallelism. It can be moved or copied anywhere on the network. Its size limit is defined by the OS.

![Federated Database](image)

**Figure 3.16: Objectivity database layout.**

Persistent objects are clustered in **containers**. This strategy allows great access performance. Indeed, objects within a container are physically placed in the same memory space. So, it is pertinent to put objects in the same container when they are often accessed together: a logical partitionning of data. A container size is defined as a number of pages; the limit is at 65530 pages. It also contains a page map which store the conversion from logical pages to physical pages. As for the access, there can be only one writer but multiple readers. It is done through the use of locks.

3.4.3 Page Modification

When an object is modified, its corresponding page is obviously modified. A new page is then written in the container but the old page is not removed. The current page map is recorded into a journal file.

After the commit, the page map is updated and stored to disk. The old page is marked as unused, the journal file is truncated and the lock is removed.

After an abort, the new page is marked as unused and the lock is removed.
3.4.4 Persistence

The persistence in Objectivity is a persistence by inheritance. This writing will focus on the C# interface of Objectivity.

The persistence designer is a plugin for Visual Studio allowing to define persistent objects. It can also create federated databases and add schema classes and collections and define relationships. It automatically generates code to reflect the modifications in the architecture.

For each class, the persistence designer will generate two files. Indeed, .NET partial classes are used to separate the application code from the persistence part. As persistence is by inheritance, both classes must inherit from ReferenceableObject. The file concerning the persistence is used to define schema class and attributes and should not be modified by hand whereas the application code can be modified easily.

Relationships

The relationships between objects must be declared within classes so that they would automatically be maintained.

- They can be unidirectional or bidirectional. In unidirectional relationships, only one class has a link to another class whereas bidirectional relationships define a relationship between two objects which know each other.

- They have a cardinality. At each side of a relationship can be defined a number; this is the number of objects which can be attached to the relationship. These are the types supported:
  - one-to-one
  - one-to-many
  - many-to-one
  - many-to-many

- They can be used to create composite objects.

The storage of the relationships can be chosen:

- Non-inline relationships are stored in the default association array. This is the default mode and takes up little space for small number of objects.

- Inline relationships are stored as an attribute which is a reference to a single object or to an array of objects. Its traversing is very efficient.

For bidirectional relationships, the storage properties must be the same for the two directions. Moreover, Objectivity makes sure that the referential integrity is respected. Adding or removing objects in a binary relationships automatically updates its inverse relationship and deleting an object automatically removes it from any relationship in which it appears.
Some basic semantics can be added for relationships (see fig. 3.17). The first possibility is the deletion propagation: when an object is deleted, all the objects linked to it by a relationship are also deleted. This is exactly what is needed for composite objects. The second possibility is the lock propagation. Locking an object lead to the locking of all the associated objects. Note that those mechanisms are optional and so, need to be explicitly activated.

Finally, Objectivity provides a basic control on the behaviour of relationships of an object which has been copied (see fig. 3.18). Here are the different strategies:

- **copy**: The new object has the same relationships as the old one.
- **drop**: The new object does not have any relationship and the old one keeps its relationships.
- **move**: The new object takes the relationships from the old one which ends without any relationship.

Also, this behaviour can be defined for each relationship of an object.
3.4.5 Transaction

Just like any other databases, interactions with them must happen inside a transaction which respects the ACID properties. If a transaction is committed, all the modifications during the transaction are saved in the federated database. But if a transaction is aborted, the modifications are not applied to the federated database which is in the same state as before the beginning of the transaction.

Objectivity libraries contain static functions to start and shutdown a database and to connect to one.

```
// Initialize Objectivity/DB
Objy.Startup();
// Establish a connection to the federation
Connection connection = Objy.GetConnection("publications.boot", true);
// Shutdown Objectivity/DB
Objy.Shutdown();
```

The first thing to do is to initialize the Objectivity. Moreover, it cannot be shut-downed if there are still some database functions further. The connection is limited to one per federation for an application process.

A transaction is managed by a session which is the link between the application and the connected federated database. So, this is the role of the session to start and to end a transaction. Several of them can be created in a thread and they can also be shared between different threads. However, they are all linked to the same connection.

A cache is associated and managed by a session. It stores all the modified objects during the transaction. If the cache size is not enough, some pages are pre-written to disk. Aborting a transaction flushes the cache but it is left untouched after a commit.

```
Objy.Startup();
Connection connection = Objy.GetConnection("publications.boot", true);
// Create a session
Session session = connection.CreateSession("main");
// Start a transaction
session.BeginTransaction(OpenMode.Update);
// Get the federation
Federation fd = session.Federation;
```
// Lookup or create the publications database
Database db;
if (fd.HasDatabase("PublicationsDB"))
{
    db = fd.LookUpDatabase("PublicationsDB");
}
else
{
    db = new Database(fd, "PublicationsDB");
}
...
session.CommitTransaction();
Objy.Shutdown();

3.4.6 Collection

Objectivity includes several persistent collections. The sets, lists and maps are implemented. These collections can either be ordered or unordered and be scalable or non-scalable (see fig. 3.2).

<table>
<thead>
<tr>
<th>Scalable</th>
<th>Non-Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ordered</strong></td>
<td></td>
</tr>
<tr>
<td>TreeListX&lt;T&gt;</td>
<td></td>
</tr>
<tr>
<td>TreeMapX&lt;K,V&gt;</td>
<td></td>
</tr>
<tr>
<td>TreeSetX&lt;T&gt;</td>
<td></td>
</tr>
<tr>
<td><strong>Unordered</strong></td>
<td></td>
</tr>
<tr>
<td>HashMapX&lt;T&gt;</td>
<td>Map&lt;T&gt;</td>
</tr>
<tr>
<td>HashSetX&lt;T&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: The different types of persistent collections.
Source: [13]

3.4.7 Iterator

An Iterator is an object which allows to filter persistent objects of a certain class based on a criteria. It is designed to work on either a particular collection, container, database or federation. The criteria is expressed as the string of a PQL predicate.

// Print all authors that have a name starting with M
Iterator<Author> iAuthors =
    new Iterator<Author>(authors, "name =~ \"M.*\"");
Author author;
while ((author = iAuthors.Next()) != null)
{
    Console.WriteLine("{0} has published {1} publications",}
Unfortunately, this method is not really efficient for looking up objects because this is the client that evaluates the predicates. This can be improved by indexing the data and limiting the scope of action of the iterator. Moreover, the client can start working on data while the result is being built.

### 3.4.8 Scope names

Scope names are a generalization of database roots which are not supported in the C# version of *Objectivity*. So, a name can assigned to an object, a container, a database or a federated database for an easier access. It is also possible to group objects under a scope name.

```csharp
// Assigning a scope name
HashSetX<Book> books = new HashSetX<Book>(db);
db.NameObject(books, "books");
// Looking up a scope name
HashSetX<Book> allBooks =
(HashSetX<Book>)(HashSetX<IReferenceableObject>) db.LookUpObject("books");
// Unassigning a scope name
db.UnnameObject(allBooks);
```

### 3.4.9 Retrieving Objects

There are different techniques to retrieve objects in *Objectivity*. Some of them have already been explained above.

The first technique is by using scope name. The second one is by going through an object graph. Indeed, when an object has a relationship or a reference to another object, it defines a graph. The search can follow this link and so going from one object to another. The third technique is by using iterators. The last technique that will be mentioned is the parallel query. As said at the beginning of the chapter, *Objectivity* provides a PQE which is able to split the target of a query between different query servers.

All these techniques mainly uses a filtering based on content to retrieve objects. Therefore, a predicate query language is included. It supports primitive types and strings.
3.5 Summary

Here is a table comparing the previous databases.

<table>
<thead>
<tr>
<th></th>
<th>db4o</th>
<th>Versant</th>
<th>ObjectStore</th>
<th>Objectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Object-based</td>
<td>Object-based</td>
<td>Page-based</td>
<td>Container-base</td>
</tr>
<tr>
<td>Persistence</td>
<td>Reachability</td>
<td>Reachability</td>
<td>Instantiation</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Java</td>
<td>Java</td>
<td></td>
<td>C++</td>
<td>C#</td>
</tr>
<tr>
<td>Queries</td>
<td>SODA</td>
<td>VQL</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Standard</td>
<td>LINQ</td>
<td>LINQ, JDO</td>
<td>JDO</td>
<td>LINQ, JDO</td>
</tr>
</tbody>
</table>

Table 3.3: Object Databases Comparison
Chapter 4

Choosing a Query Language

After reviewing some of the most popular object databases, time has come to determine a query language which has the best chances to become a standard. The old standard OQL is not supported by any of these databases. Each one of them has developed their own mechanisms to make queries, so they are not compatible.

However, many object databases manufacturers have added support for some recent query libraries such as LINQ and JDO and have put highlight on them. They have been developed without any particular object database in mind but as a standard. That is why they are completely independant from the underlying database. Even if the opinions of manufacturers on a standard adoption are pretty different, these so-called standards are getting more and more adopted. A standard can become a real standard only if it is adopted.

Among the ideas for a standard, an interesting one would be the ability to query not only object databases but also the classical relational databases. That way, learning one language could be enough and that would facilitate life of users. Moreover, more people would try using an object database instead of their usual relational database. If the language can look syntactically like SQL which is generally first learned, the switch would be even smoother.

There are three directions possible in the development of the standard:

- API with its own query language provides methods to make queries: JPA, JDO.
- The query capabilities are integrated in the programming language: LINQ.
- A new query language is developed as a programming language: SBQL.

In the following sections, each one of them will be introduced and the basic usage of their query language will be described. However, this writing will not focus on how to make them work with a database. Advantages and inconvenients will be revealed to see what is the better candidate for being the standard of the future.

\[1\]In this chapter, the basic usage and syntax of SQL is supposed known.
4.1 JPA

The Java Persistence API (JPA) is a Java programming language interface which manages object persistence. In fact, it was developed to work with relational databases and to be an independent Object Relational Mapper. But it is usable with db4o databases by using DataNucleus Access Platform. The API is defined in the javax.persistence package.

The query language for JPA is called Java Persistence Query Language (JPQL).

4.1.1 Queries

The queries are defined in a string as :

```
SELECT [<result>]
    [FROM <class(es)>]
    [WHERE <filter>]
    [GROUP BY <grouping>]
    [HAVING <having>]
    [ORDER BY <ordering>]
```

The syntax is similar to SQL. To create a query, the EntityManager.createQuery is called with the query string as an argument.

```
Query q = em.createQuery("SELECT p FROM Person p
    WHERE p.lastName = 'Jones'"IŞ;
```

em is an EntityManager which is the interface which interacts with the database. They can be created using the EntityManagerFactory.createEntityManager() method. To get results, it must be executed. There are two possibilities :

- If the result is 0 or 1 object, the getSingleResult() method is used.
- If the number of objects returned is higher than 1 or unknown, the getResultList() is used.

Using Java-like syntax, relations between objects can be directly used.

```
SELECT c.address.street FROM Company c WHERE c.name = 'Random House'
```

The queries in JPQL are all polymorphic but their domain can be restrained making them non-polymorphic or partially polymorphic. TYPE() can be used for that purpose. In this example, Magazine is a subclass of Tabloid. It selects all the magazines which are tabloids.

```
SELECT x FROM Magazine WHERE TYPE(x) = Tabloid
```

To easily reuse some queries but with different values, they can be parametrized. The parameters can be either named or numbered. However, only one type of parameter can be used in a same query. To set values, the setParameter() method is provided. The named parameters are prefixed by a colon (":".

\[2\]For more details on the usage, see http://openjpa.apache.org/builds/latest/docs/manual/jpa_overview_emfactory.html
Query q = em.createQuery("SELECT p FROM Person p
        WHERE p.lastName = :lastname AND p.firstName = :firstname");
q.setParameter("lastname", "Sham");
q.setParameter("firstname", "Shamoko");

The numbered parameters are prefixed by a question mark ("?"), They must be numbered starting from 1, so they depend on the position in the query.

Query q = em.createQuery("SELECT p FROM Person p
        WHERE p.lastName = ?1 AND p.firstName = ?2");
q.setParameter(1, "Sham");
q.setParameter(2, "Shamoko");

Finally, JPQL provides some pre-defined functions:

- CONCAT(string1, string2)
- SUBSTRING(string, startIndex, [length])
- LOWER(string)
- UPPER(string)
- LENGTH(string)
- LOCATE(searchString, candidateString [, startIndex])
- ABS(number)
- SQRT(number)
- MOD(number, divisor)
- INDEX(identification_variable)

Here are the syntax for update and delete queries.

DELETE FROM [<class>] [WHERE <filter>]
UPDATE [<class>] SET item1=value1, item2=value2
[WHERE <filter>]

To execute these queries, the executeUpdate() method is called on them. It returns in the first case, the number of objects deleted and in the second case, the number of objects modified. JPQL supports several aggregation functions:

- MIN
- MAX
- AVG
- COUNT
4.2 JDO[15]

Another Java standard for object persistence is Java Data Objects (JDO). Contrary to JPA, it was designed to be compatible with any type of database. So, the interface to persist Java objects would be the same whether the type of storage is a relational database, object database or XML. This standard would make possible to switch code developed for one database to be compatible with another JDO supported database without changing it. It is supported by Versant Object Database, ObjectStore and Objectivity. The API is defined in the javax.jdo package.[18] JDO introduces its own OO query language JDOQL. The basic usage will be exposed.

4.2.1 Queries[19]

JDO provides in fact two different means to express JDOQL queries:

- Using a Single-String JDOQL (the new way, introduced in JDO 2).
- Using the Declarative JDOQL (the old way).

First, here is an example of the same query using the two possibilities.

Declarative JDOQL:
Query q = pm.newQuery(mydomain.Person.class);
q.setFilter("lastName == "Jones" && age < age_limit");
q.declareParameters("double age_limit");
List results = (List)q.execute(20.0);

Single-String JDOQL:
Query q = pm.newQuery("SELECT FROM mydomain.Person
WHERE lastName == "Jones" + " && age < :age_limit PARAMETERS double age_limit");
List results = (List)q.execute(20.0);

Before explaining the details of the syntax, the two different approaches are pretty obvious. Indeed, in the Single-String JDOQL, as the name implies, everything is expressed in the String whereas, in the Declarative JDOQL, different mutator methods are used to declare the parameter and define the filtering. However, the arguments of those methods are still string.

pm is a PersistenceManager which is pretty much the equivalent of EntityManager in JPA. It allows interactions with the database and contains a single transaction. It provides the newQuery() method which creates queries. This can take the Single-String query as an argument or the class of an object. In the second case, this method can also have a second argument which is the filtering. So, the first two lines of code in the previous example can be changed to:

Query q = pm.newQuery(mydomain.Person.class,
   "lastName == "Jones" && age < age_limit");
Calling the `execute()` method with the values of parameters as arguments simply execute the query. More details on what it returns will be given further. Up to three parameters can be given a value with this method. Otherwise, the `executeWithArray()` method should be used. It takes an array of parameter values as an argument.

The Single-String JDOQL will be focused on because this is the most recent and its syntax is similar to SQL:

```sql
SELECT [UNIQUE] [<result>] [INTO <result-class>]
    [FROM <candidate-class> [EXCLUDE SUBCLASSES]]
    [WHERE <filter>]
    [VARIABLES <variable declarations>]
    [PARAMETERS <parameter declarations>]
    [<import declarations>]
    [GROUP BY <grouping>]
    [ORDER BY <ordering>]
    [RANGE <start>, <end>]
```

The pattern is pretty similar to JPQL. But some new keywords have been added. The old ones are used the same way as in JPA, so they do not need any explanation.

**UNIQUE** The query returns the first result and not the entire collection. It returns `null` if the size of the result is 0.

**<result>** The result defined in the `SELECT` clause can be either `this`, a field name, a variable, a parameter, an aggregate, an expression involving a field or a navigational expression. If it is empty, the objects returned will be of the type of the class defined in the `FROM` clause. This is more powerful than the one in JPQL. Besides, it defines the type returned by the `execute()` method:

- **Object** The result consists of a single row and a single column.
- **Object[]** The result consists of a single row but multiple columns.
- **List<Object>** The result consists of a single column and not only aggregates.
- **List<Object[]>** The result consists of multiple columns and not only aggregates.

**INTO <result-class>** By default, each element in the result is of the candidate class. This behaviour can be changed by choosing a `result-class`. Each element will then be of the `result-class` class.

**EXCLUDE SUBCLASSES** JDOQL queries are polymorphic. By using this keyword, objects which are a subclass of the candidate class will not be selected. This cancels the polymorphism of the query.

**VARIABLES** It is used to declare variables which connect two parts of a query. The following query selects the suppliers who have a product called "Beans".
Query query = pm.newQuery("SELECT FROM mydomain.Supplier " + "WHERE products.contains(prod) && prod.name == "Beans" " + "VARIABLES mydomain.Product prod");

The variable declaration (explicit variable) allows user to choose its type. But the type can also be determined by the query compiler so, it is not obligatory to declare them (implicit variable).

**PARAMETERS** The parameters can be declared via this keyword (explicit parameter). But they can also be defined by prefixing its name with a colon character (":").

**Explicit Parameter :**
Query query = pm.newQuery("SELECT FROM mydomain.Product WHERE price < limit " + "PARAMETERS double limit");

**Implicit Parameter :**
Query query = pm.newQuery("SELECT FROM mydomain.Product WHERE price < :limit");

**<import declarations>** They work just as programming in Java. Sometimes, parameters and variables come from a different class than the candidate class. The complete name of the classes can be used or the different classes can be imported.

**RANGE <start>, <end>** It simply sets the range of results to return.

For the filter part of the queries, JDOQL provides some built-in methods, which can be useful:

- **String methods :**
  - startsWith(String)
  - endsWith(String)
  - indexOf(String)
  - indexOf(String, int)
  - substring(int)
  - substring(int, int)
  - toLowerCase()
  - toUpperCase()
  - matches(String pattern)

3These methods are pretty standard. For the explanation of each method, see [http://db.apache.org/jdo/jdoql_methods.html](http://db.apache.org/jdo/jdoql_methods.html)
• Collection methods :
  – isEmpty()
  – contains(value)
  – size()

• List methods :
  – get(position)

• Map methods :
  – isEmpty()
  – containsKey(key)
  – containsValue(value)
  – get(key)
  – size()

• Other methods :
  – Math.abs(number)
  – Math.sqrt(number)
  – JDOHelper.getObjectId(object)

JDOQL also provides some aggregate functions :

• MIN
• MAX
• AVG
• COUNT
• SUM

Here is an example of aggregate with a grouping. The query selects the average ages of the people with the same firstname.

Query q = pm.newQuery("\select avg(age) "
  + "from mydomain.Person group by firstName");

A Java keyword which can be useful is instanceof. It is supported by JDOQL and it allows to determine if an object is of a specific type.

To finish, JDOQL supports In-memory queries. It means that queries can be made to in-memory collection of persistent objects. The collection is set via the setCandidates method.
Query q = pm.newQuery ("select from "+ "mydomain.Person where firstname = \"John\"\")
q.setCandidates (allPeople);

allPeople is a collection of People objects. The query is done in the collection
instead of the database.

JDOQL also supports subqueries. Sometimes, it is necessary for a query to use
results from other queries. Subqueries allows to do that in a single query.

SELECT FROM org.datanucleus.Employee WHERE salary >
(SELECT avg(salary) FROM org.datanucleus.Employee e)

4.3 JDO vs JPA

JDO and JPA are two standards for object persistence in Java. As seen previously,
JDO provides a query language more powerful than the JPA one. However, JDO has
been developped before JPA which was designed only for relational databases.
JDO design assures a better interoperability between the different databases and
even has a more complete ORM definition. Then why has JPA been developped when
there was already a standardization for Java object persistence? It is simply because
at that time, RDBMS vendors were not enthousiast about the idea to have one API
usable on any databases. Their solution was JPA. That is why these two standards
exist instead of only JDO.\[17\]

Unfortunately, they have the same problem. The queries are defined as a string
which is a bad point.\[8\] Indeed, because they are strings, the compiler cannot check
errors in the queries. The program must be running to see if anything goes wrong.
It there is a typo or worse, if the query does not make any sense, there are no means
to alert the user before launching the program. Besides, automatic refactoring is
also impossible. It is really a pity when agile development is pretty popular these
days.

4.4 LINQ[22]

The Language Integrated Queries (LINQ) is a part of the Microsoft .NET frame-
work (System.Linq). It adds query capabilities to .NET languages. In other words,
they are integrated into the programming language. And because of that, the queries
can directly use all the methods available in this language and their syntax can be
checked during the compilation. Its goal is to offer a standard way to access all kinds
of data sources. Therefore, some LINQ providers have been developped : LINQ to
Objects, LINQ to SQL, LINQ to XML and LINQ to DataSet. Besides, LINQ provides
an interface which can be implemented by the developpers to add support for their
own data storage. It allows to add new domain-specific operators to the standard
query operators and even redefine them. So, db4o, Objectivity and Versant have
developped their LINQ providers.
4.4.1 Queries

To ease the transition from relational databases, the syntax of the queries has some similarities with SQL. This is the declarative way based on **query expression** where **names** is an array of string. The query selects and orders the strings which are changed to uppercase in **names** and which have a length of 5.

```csharp
string[] names = { "Burke", "Connor", "Frank", "Everett", "Albert", "George", "Harris", "David" };

IEnumerable<string> query = from s in names
    where s.Length == 5
    orderby s
    select s.ToUpper();
```

This example can also be written as a **method-based query**.

```csharp
IEnumerable<string> query = names
    .Where(s => s.Length == 5)
    .OrderBy(s => s)
    .Select(s => s.ToUpper());
```

The query expressions are shorter and easier to read. The compiler automatically converts them to method-based queries.

The methods **Where**, **OrderBy** and **Select** are extension methods. They allow to add methods to existing types without directly modifying the code of these types. Besides, there is no difference in calling one of their methods or an extension method.

The different arguments are **lambda expressions** which are fragments of code.

Another important concept in LINQ is the expression tree. In the example above, the query expression returns an **IEnumerable<string>** (more generally an **IEnumerable<T>** if the type is unknown). The particularity of this interface is the possibility to use the **foreach** statement on a collection which implements it. In this case, there is no expression tree. This is not needed.

When using LINQ to SQL for example, the query expression returns an **IQueryable<T>** which is used to hold an expression tree. This is simply a representation of the code for the query. As with any tree, it is possible to explore it. So it is easier to translate code by analyzing the tree than by analyzing an instruction line. In the case of LINQ to SQL, the data structure must be transformed into SQL query and then sent to the database server.

In the example, the query is not yet evaluated. This is only done when it is iterated over using **foreach** or doing it manually. This is called the **Deferred Query Evaluation**.

---


foreach (string item in query)
        Console.WriteLine(item);
    // BURKE
    // DAVID
    // FRANK
    names[0] = "Bob";
foreach (string item in query)
        Console.WriteLine(item);
    // DAVID
    // FRANK

In fact, the query is executed each time the variable `query` is iterated over. So, if the variable `names` is modified, the same `foreach` statement can return other values. The query is reused. However, some operators can force the query evaluation, such as `ToArray` and `ToList`. The results are then cached. This time, the `foreach` statement will always return the same values even if the variable `names` is modified.

```csharp
var queryArray = query.ToArray();
foreach (string item in queryArray)
        Console.WriteLine(item);
    // BURKE
    // DAVID
    // FRANK
    names[0] = "Bob";
foreach (string item in queryArray)
        Console.WriteLine(item);
    // DAVID
    // FRANK

The structure of a query expression can be defined using the following grammar taken from [22].

```csharp
query-expression ::= from-clause query-body

query-body ::= query-body-clause* final-query-clause query-continuation?

query-body-clause ::= (from-clause
    | join-clause
    | let-clause
    | where-clause
    | orderby-clause)

from-clause ::= from itemName in srcExpr
```
join-clause ::= join itemName in srcExpr on keyExpr equals keyExpr
   (into itemName)?

let-clause ::= let itemName = selExpr

where-clause ::= where predExpr

orderby-clause ::= orderby (keyExpr (ascending | descending)?)*

final-query-clause ::= (select-clause | groupby-clause)

select-clause ::= select selExpr

groupby-clause ::= group selExpr by keyExpr

query-continuation ::= into itemName query-body

In fact, the different clauses do not need any explanation because their meaning are
the same as in SQL. The only exception is the let clause which allows to define a
name for a result of an expression.
A query expression always begins with a from and ends with a select or groupby
clause. The first from clause is followed by zero or more from, let, join, where
and orderby clauses.
Although their overall meanings are the same, they can have some subtleties.
For example, in addition to select, a selectmany keyword exists. The select
keyword returns a value for each element in the source sequence. If the value is a
sequence, the user must use two foreach. The use of selectmany allows to expand
the returned sequence.

string[] text = { "Albert was here",
   "Burke slept late",
   "Connor is happy" };

// select the splitted strings
var tokens = text.Select(s => s.Split(' '));
// each element is a string[]
foreach (string[] line in tokens)
   foreach (string token in line)
      Console.Write("{0}.", token);
// Albert.was.here.Burke.slept.late.Connor.is.happy.

var tokens = text.SelectMany(s => s.Split(' '));
// each element is a string
foreach (string token in tokens)
   Console.Write("{0}.", token);
// Albert.was.here.Burke.slept.late.Connor.is.happy.
LINQ provides other operators which are expressed as method calls such as but not limited to:

- **Aggregate Operators**:
  - Count
  - Sum
  - Min
  - Max
  - Average
  - Aggregate

- **Quantifiers**:
  - Any
  - All

- **Element Operators**:
  - First
  - FirstOrDefault
  - ElementAt

The greatest strength of LINQ is also its biggest weakness: the integration in the programming language. This particularity allows to avoid the problems in JDO and JPA which involve query string. This time, the queries are not defined as string, they are a part of the programming language. So they can access the methods of the objects directly. Moreover, the compiler can now detect syntax errors and checks if the field of an object defined in the query really exists. Of course, the automatic refactoring is now possible. Changing the name of a field for an object in the database, for example, do not involve manual code editing anymore. Using LINQ also facilitates the debugging of the queries.

However, the .NET language was not designed as a query language. It has been patched to add query capabilities. It makes LINQ design more complicated. And eventually, the programming language will be the limiting factor of the evolution of LINQ.\[5\]

### 4.5 SBQL\[32, 1\]

The query language SBQL is based on the **Stack-Based Architecture (SBA)**.

The Stack-Based Architecture (SBA) is a formal methodology addressing object-oriented database query and programming languages.\[32\]
The query language is seen not as an extension of programming language but as a programming language itself. So, a theory was born to mix the two concepts. This is SBA. Its goal is to provide a uniformity for the queries and the programs using these queries.

The idea is to define the data structures involved in query processing and then express the meaning of the operators as actions on these structures. This way to define the semantics is called abstract implementation. SBA provides three structures: an object store, an environment stack and a query result stack.

It is pretty similar to the way programming language operates on runtime except that the stacks treat indifferently single data and collections. The stack and the object store are separated; the stack do not contain the data but only references to data. Besides, expressions and queries are not differentiated. They can all be used as arguments of statements, as parameters of procedures, functions or methods and as a return from a functional procedure.

SBQL plays the same role as relational algebra for the relational model, but SBQL is incomparably more powerful.

SBQL has the capabilities to work with a wide range of data structures whether it be querying or manipulating data. It is precise thanks to the semantics specification and was designed as a programming language. It includes several classical concepts such as functions, types, classes, methods, ... and respects the common programming language principles.

Because SBA and SBQL work with data structures instead of data models, they are completely independent of the database models. The data model concepts simply need to be translated as data structures. So, a part of SBQL queries are defined to manage these data structures with full algorithmic power. For example, SBQL clearly knows how to query a object database or a relational database depending on the data structure it manipulates.

Here is a list of 10 qualities taken from SBQL:

1. Orthogonal syntax, full compositionality of queries.
2. Universal formal semantics based on abstract implementation.
3. Computational universality, advanced data structures, integration with PL constructs.
4. Strong typing of advanced O-O queries and programs.
5. Several advanced implementations, further ones pending.
6. Fully transparent O-O virtual updatable views.
7. Strong potential for query optimization.
8. All O-O notions treated formally and uniformly.
9. Sound and manageable metamodel.
10. The potential for distributed query processing.
To illustrate the syntax of SBQL, here are some examples:

Get the average number of employees in all departments:
\[
\text{avg(Dept.count(employs))}
\]

For each person having no salary give the minimal salary in his/her department:
\[
\text{for each (Emp where not exists(sal)) as e do}
\text{e.changeSal( min(e.works_in.Dept.employs.Emp.sal) )}
\]

The queries are pretty compact and seem to flow naturally.

4.6 The Standard of Tomorrow

To finish, SBQL and LINQ will be compared to see which one has the most chances to become the standard of tomorrow.

First, LINQ object model is not as rich as SBQL model and LINQ queries are more complex than in SBQL.

Get departments together with the average salaries of their employees

SBQL: Department join avg(employs.Employee.salary)

LINQ: from d in Department select new {
  dpt = d,
  avg = (from e in d.employs select e.salary).Average()}

Indeed, they are not uniform: they mix SQL clauses, lambda expressions and method calls which make the query more difficult to understand. Moreover, the usage of auxiliary variable names are optional in SBQL.

LINQ is a .NET framework, it is an extension of the programming language. To make native LINQ queries in Java, their syntax would need to be changed because C# and Java do not work the same way. They are different languages after all.

The design of LINQ do not separate clearly the front-end and the back-end. That is why there are a lot of different providers for LINQ (LINQ to Objects, LINQ to SQL, LINQ to XML, ...) and a lot more will come.

In LINQ, the programming expressions and queries are differentiated. The queries cannot be used as left-hand parameter.

In fact, SBQL do not have any of these problems because this is a programming language designed to make queries which also include programming expressions.

Unfortunately for SBQL, reality can be harsh. Indeed, despite its superiority, it is not supported by any of the big object database vendors. For now, they have chosen LINQ and the most of them are convinced that it will be the standard adopted by everyone. Moreover, they do not seem to absolutely want a standard that works with every programming languages. They also support JDO which is only for Java.

Some even say that the standard for object persistence in Java should be inspired by LINQ: A query language integrated within the programming language.
Choosing a Database

Now that a standard has been chosen, it is the turn of the database. Before, the usage context must be defined.

As said in the introduction, the database will be used as a first contact with object database for the students. So, only the basic functionalities are needed and the performance is not the major point. The database server and the client will run on the same machine. Finally, the database must obviously support the LINQ which is what the students would learn.

After reviewing 4 different databases, the major difference between them can be noted (see table 3.3 page 48). Indeed, the architecture can be either container-based, page-based or object-based. This choice will greatly influence the concurrency model, the network model, query implementation and thus the performance. It is important to carefully define for what type of application the database is needed. Each database has its strengths and weaknesses, this is even more true for object database. However, as stated above, it is not the main point in this use case.

Among the 4 databases, 3 support LINQ: db4o, Versant Object Database and Objectivity. Only the first one is open source whereas the others are commercial solutions. So, db4o is a good solution for the university which tries to move towards the open source world. The students can also download and install it legally. Moreover, it provides the embedded mode (see section 3.1.5) which allows to have the application and the database on the same machine easily without the need to configure a virtual server. As they need to be installed on each computer, this feature can save time.

Because LINQ is a part of .NET, the machine must have Windows. This is not entirely true. As of now, LINQ is supported in Mono which is an open source .NET framework. That solution makes LINQ available on other platforms such as Linux. db4o can work with Mono but not directly, it needs to be patched. There is a project which provide patched version of db4o. However, there do not seem to be any update, only an old version is provided.

For indication only, the figure 5.1 shows the performance of db4o with some relational databases using the PolePosition benchmark with the Complex circuit. It shows the performance when writing, reading, querying and updating an object graph which is composed of several classes with an inheritance of 5 levels. The performance of db4o is pretty good compared to the others. Hibernate is an
object-relational mapper so it is logical that its performance is worse than an object
database which do not care about the object-relational mismatch.

In conclusion, db4o is the object database most adapted for this use case.
Chapter 6

Development of a LINQ-like

LINQ is limited to the .NET languages. The idea here is to lift this limitation by developing a LINQ-like, a library which would add type safe language integrated query capabilities to another programming language. Rather than starting from scratch, some researches have been done to see if there were already some libraries like that.

6.1 Pynq[29]

Python Language Integrated Query (pynq) is as its name implies the version for Python. It implements the building of Expression Tree which is a theory used in LINQ.

Because Python is a non typed and interpreted programming language, no type checking can be done. This library only makes querying collections easier. Just like LINQ, it is also possible to define other providers.

6.2 lambdaj[21]

lambdaj has been developed for the manipulating of collections in Java. It allows to iterate collections without writing explicitly the loops to make the code more readable. So, it provides some of the most useful functionalities when iterating collections is needed :

- filtering
- converting
- extracting a property
- sorting
- grouping
- calling a method on each item
- aggregating
• concatenating string

At the time, closure was not included in Java. Therefore, lambdaj implemented a pseudo-closure. Here is an example of code with lambdaj and its equivalent in plain Java.

// lambdaj
List<Person> sortedByAgePersons = sort(persons, on(Person.class).getAge());

// plain java
List<Person> sortedByAgePersons = new ArrayList<Person>(persons);
Collections.sort(sortedByAgePersons, new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return Integer.valueOf(p1.getAge()).compareTo(p2.getAge());
    }
});

It seems obvious that the first code is more readable and easier to maintain than the second one.
In fact, lambdaj is not really a LINQ-like. The single common point is that the collection querying is made easier but lambdaj does not use the theory of Expression Tree.

6.3 JaQue[14]

Another solution in Java is the Java integrated query (JaQue) library. It provides LINQ-like query capabilities by building an expression tree using ASM\(^1\). This is the equivalent of LINQ but with the Java syntax. The Java can now have access to this powerful library for querying. There are some providers available: JaQue to Objects and JaQue to XML. Moreover, JaQue to JPA is under development.

Closures have been introduced in the version 7 of Java and the library is already compatible. In the future, its syntax will be changed to take into account the new syntax.

Iterable<? extends Number> r = from(data,
    where( { Integer i => i > 5 },
    skip(10,
        select( { Integer i => 4 } ))));

The current syntax is less clear :

for (int i : where(nums, { int t => t < 2 }))
    System.out.print(i);

JaQue is currently the most advanced LINQ-like for Java that was found.

\(^1\)ASM is Java bytecode manipulation and analysis framework. See [http://asm.ow2.org/](http://asm.ow2.org/)
6.4 Developing a JDO provider for JaQue

As JaQue already exists, it would be a waste to try to develop another one for Java. So, to contribute to its development, a JDO provider can be a good idea. Remember that contrary to JPA, JDO is designed to work with all kinds of databases.

Another possibility would be to develop a provider for a database such as ObjectStore. But this would limit its usage, it would be necessary to have a provider for each existing database. As JDO is already supported by several of them, only one provider, JaQue to JDO, would be compatible with them. However, it adds another layer on top of JDO, so the performance would certainly suffer. But it is a good beginning, if JaQue is more and more adopted, the database vendors will certainly develop a provider for their products without relying on JDO.

The three main classes in JaQue are:

- **Query**: The class provides methods for receiving and returning object streams. This is here that are defined the `join`, `where`, `select`, ... They can be nested.

- **Expression**: The class provides some rules to be used with Query class: `add`, `subtract`, `greaterThan`, `lessThanOrEqual`, ...

- **Operation**: The class provides operations which to enumerate the object stream: `count`, `first`, `last`, ...

A JPA provider is available. As JPA is pretty similar to JDO, the provider will be the inspiration for the development of the new one. Here is an example for the JPA provider.

```java
EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("hibernate");
EntityManager em = emf.createEntityManager();
JaQueEntityManager jem = new JaQueEntityManager(em);
Query<Order> orders = jem.from(Order.class);

System.out.println(count(where(new Predicate<Order>() {
    public Boolean invoke(Order t) throws Throwable {
        return t.getOrderID() > 11000;
    }
})), orders));
```

In JDO, that would give:

```java
PersistenceManagerFactory pmf = JDOHelper
    .getPersistenceManagerFactory("datanucleus.properties");
PersistenceManager pm = pmf.getPersistenceManager();
JaQuePersistenceManager jem = new JaQuePersistenceManager(pm);
Queryable<Order> orders = jem.from(Order.class);

System.out.println(count(where(new Predicate<Order>() {
    public Boolean invoke(Order t) throws Throwable {
        return t.getOrderID() > 11000;
    }
})), orders));
```
return t.getOrderID() > 11000;

Some adaptations in the code have been made to work with JDO. For now, only select and where are supported. The aggregates do not work yet. Despite much effort put into finding the solution to this problem, it has not been solved so far. This code can be continued to add more features.
Chapter 7

Conclusion

The main goal of this thesis was to determine if object databases are a technology that would still live in the future and what new standard should be learned. Indeed, they are still introduced at Université Libre de Bruxelles but the old standard adopted by ODMG (OQL and ODL) is old and not used in practice.

The starting point was the site http://odbms.org which provides all kinds of documents concerning ODBMS and some coming from International Conference on Objects and Databases. The research has been limited to documents after 2000 to stay up-to-date enough. With all this material, it was now possible to dig into object databases and reveal its power.

The first step was to define the concept of object database and to expose the advantages in comparison with the most used DBMS, the relational database. The main inconvenient was the impedance mismatch between object oriented data coming from the application and the relational data in the database. Object database was born from the wish to combine object-oriented concepts and database concepts: To be able to save the state of an object directly into the database without the need to make any performance-hungry conversion.

Even though they are less known than their relational counterpart, they are used in various domains that require to manipulate complex bits of information without losing in performance. Unfortunately, they seem more complex to use for everyone because of the absence of an adopted standard like SQL is for the relational database.

To show how object databases work, four of the most popular ones (based on their participation in ICOOB) was reviewed and had their specificities described. They offered the same basic functionalities but they implemented them completely depending on the architecture adopted. So, the performance can vary greatly according to the usage that the database is needed for. Each one has its strengths and its weaknesses. Among the four databases, three architectures appear: container-based, page-based and object-based. The choice in the architecture has an impact on the concurrency model, the network model and the query implementation. Compared to relational databases, the choice of an object database is even more essential to have the best performances.

Rapidly, it was clear that a standardized query language was not a priority; each database vendor provide their own way to query the stored objects. However,
in the recent years, they decided to include the support of some standards such as JDO for Java and LINQ for .NET. They considered them as big advances in the standardization of object databases which never really happened before.

That is why JPA, JDO, LINQ and SBQL had their basic usage and syntax described. JPA and JDO are two standards for object persistence in Java and they have the same flaw. The queries are expressed as strings and so cannot take advantage of the type checking and the refactorings contrary to the two others. SBQL is a programming language thought for querying and is therefore the most powerful. However, LINQ was chosen because it is already supported by several object database vendors and is integrated into the programming language. Besides, LINQ is part of the .NET language which is the most popular with Java. In view of the facilities brought by LINQ for querying collections and any types of database, its future is nearly assured, especially with the help of its creator, Microsoft.

Finally, the last step was the choice of the database to install. Indeed, learning to use LINQ on paper is not very interesting. Using it, querying a database with it and seeing that it really works have much more value. The best choice among the four described in this writing for being installed at the University is db4o. It is open-source, can be used locally and of course, it supports LINQ.

The bonus part of this thesis covered the development of a LINQ-like for Java or rather a provider for JaQue. The success and the possibilities of LINQ tempted people to integrate querying capabilities into their favorite language. In Python, Pynq implements the building of expression tree and lambdaj provides in Java facilities for manipulating collections. The most advanced LINQ-like in Java that was found, is JaQue. The idea was to develop a provider, JaQue to JDO. Instead of making a provider for a given database, this decision was made so that it would be compatible with any other JDO-supported database and so, be available for more people. For now, it supports only select and where.

The Future

For the standard LINQ, only the future will tell us if it was the right choice. For now, based on all the documents found, the conclusion is clearly LINQ. Maybe in five years, things will be different and another thesis will be required to analyze the market of the object databases.

The JaQue to JDO provider only supports select and where. The support for aggregates has begun but is still unfinished. This development can be continued to make a full-features a complete provider.
List of Figures

3.1 Embedded Mode : Direct File Access .................................. 15
3.2 Embedded Mode : Client Session ....................................... 15
3.3 Client/Server Mode : Networking Mode ............................... 15
3.4 Client/Server Mode : Out-of-Band Signalling ....................... 16
3.5 Versant architecture .................................................... 19
3.6 Versant dual cache architecture ...................................... 20
3.7 Versant multi-threaded architecture ................................. 20
3.8 Versant JVI architecture ............................................... 21
3.9 ObjectStore architecture .............................................. 30
3.10 Page fetching ........................................................... 32
3.11 A classical scenario of permit revocation ........................... 33
3.12 A example of configuration with different servers and clients running on different platforms ........................................... 34
3.13 The database and the segments ....................................... 35
3.14 The ObjectStore collections .......................................... 39
3.15 Objectivity architecture .............................................. 41
3.16 Objectivity database layout ......................................... 42
3.17 Relationship propagation ............................................. 44
3.18 The consequences of an object copy with one relationship .......... 44
3.19 One connection, several sessions and several threads .................. 45

5.1 db4o benchmarks ....................................................... 64
List of Tables

2.1 Databases comparison. ................................................. 6
3.1 The combination of permits and locks. ............................. 33
3.2 The different types of persistent collections. ....................... 46
3.3 Object Databases Comparison ...................................... 48
Bibliography


