TUHH
Technische Universität Hamburg-Harburg

STS
Software Technology Systems

Type-safe Functional Queries over Object-oriented Data Models

Diplom Thesis
submitted by
Khaled Alshurafa

supervised by
Prof. Dr. Ralf Möller
Prof. Dr. Dieter Gollmann

Hamburg University of Science and Technology
Software Systems Institute (STS)

July 30, 2009

Abstract

Utilizing the Java Persistence API (JPA) and its query language Java Persistence Query Language (JPQL), developers can easily interact with relational data structures from the object level. JPQL is overshadowed however with the lack of proper type-safe and functional capabilities. Unlike its counterpart on the .NET framework, Language INtegrated Query (LINQ). This work discusses, on the one hand, possibilities of translating LINQ into JPQL, and the impact of the newer JPA on the research done so far, and on the other hand, processing of query results as secondary-memory objects that are joined with other ones operating on the main-memory. Large data sets cannot be handled adequately in main-memory, and afflicted with mapping and memory-architecture constraints when moving from one memory to another. Simple iteration techniques in imperative languages lead typically to ineffective nested loops. To the contrary of functional languages, list comprehensions are often deployed there to handle data sets succinctly and dynamically. Comprehensions are reinforced by a proper semantic foundation.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, July 30, 2009
Khaled Alshurafa

I owe special gratitude to Prof. Dr. Möller for providing me with valuable insights and practical guidance, as well as encouragement throughout the research.

I would like to thank my family for their unwavering support, and my wife and best friend *Berrin* for believing in me.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

BNF Backus-Naur Form

DBMS Database Management System

DSL Domain Specific Language

EJB Enterprise JavaBean

EMF Eclipse Modeling Framework

IDE Integrated Development Environment

J2EE Java 2 Platform, Enterprise Edition

J2SE Java 2 Platform, Standard Edition

JAXB Java Architecture for XML Binding

JDBC Java Database Connectivity

JDK Java Development Kit

JPA Java Persistence API

JPQL Java Persistence Query Language

JSR Java Specification Request

JVM Java Virtual Machine

LINQ Language INtegrated Query

OCL Object Constraint Language

OODBMS Object-Oriented Database Management System

OQL Object Query Language

ORM Object-Relational Mapping

RDB Relational Database

RDBMS Relational Database Management System

SQO Standard Query Operators

Chapter 1

Introduction

1.1 Motivation

To reduce the complexity of accessing and integrating information, the .NET framework introduced Language INtegrated Query (LINQ) as another component of its utility belt. Thereby, it is possible to deploy .NET languages to navigate over different data models with one query language. By considering a relational back-end data store, the Java Persistence API (JPA) can be seen as its counterpart on the Java platform. It defines a standard for object-relational mapping (ORM), bringing along its own query language, the Java Persistence Query Language (JPQL). Whilst both enable programmers to directly interact with data structures from the object level, JPQL in its current version is rather simple, LINQ is robuster and above all, functional.

LINQ or JPA? Unlike JPA, LINQ offers the benefits of both compile time checking and dynamic query composition. Apart from the functionality, non-commercial solutions will be always of interest for academia. Besides, Java is the most popular programming language today[1]. It still lacks functional traits, however. Adding such qualities would help indeed in leveraging the querying capabilities of JPA. The question cannot be answered on the spot, but to make it quick, the choice has been narrowed down to handle JPA. The problem list within the JPA is reasonably bigger and worth to be analyzed. A set of working solution concepts can be revised, and eventually thrust their way into main-stream programming.

On a broader view, today's trend is to be less dependent on the relational database management systems (RDBMS), which are the most predominant database systems [CB04]. Simply due to the fact that data nowadays are more complex than what RDBMS are originally developed for. The relational model is advanced by prudent dependence on the relational algebra. With the help of ORM mechanisms, the object-oriented (OO) data modeling unifies application and database development by taking on a burden of appropriately supporting other models with prominent OO abstraction. Nevertheless, substantial denotational semantics are still missing in most OO programming and query languages. Moreover, data transport represents a major concern in computer science and especially in database structures. It is not dependent upon the use of a specific query language or data model, appearing disguised as part of other problems. Nevertheless, a look at the origins is worthwhile, which might lead to simple interpretations.

[1] The Guardian, March 2009: http://www.guardian.co.uk/technology/2009/mar/19/java-programming-software

1.2 Problem Description

The current JPA version supplies the Java community with querying capabilities of persistent relational data, with flaws however. The main concern is how JPQL offers its users blank checks to write queries and send them without further ado for evaluation. The task of verifying the syntactic and semantic correctness is left for users. The same procedure undergoes several automated steps with LINQ, but more secure and mostly more effective.

Query processing on the syntax surface might lead to sort out the problem, but will surely result in either double processing procedure or having to transfer the data from one place to another. Double processing means going over the same data structure in two or more runs. Instead, trees are walked and their data accumulated in memory are extracted once, before moving or deleting them. Processes having round-trips are noted to be simpler but slow and ineffective. Generally, evaluating and manipulating the user input on an interface would be tedious, and is for sure not a good abstraction. Such an input can be seen simply as syntactical elements describing paths, which can be later classified in Java as objects belonging to data structures. This is the basic idea used in algorithms responsible for transforming the surface syntax effectively and systematically to an abstract syntax, which is comparable to the procedure of rewriting queries before selecting, and eventually before evaluating the combinations.

Delegating the eminent features of LINQ to JPA would enable Java developers to perform qualitative queries without having to switch to another environment. LINQ-to-JPQL automation requires a semantic basis and is tainted with a number of technical restrictions, as in the case of any translation. Best practices achieved in this area could help to a certain extent. Alternatively, instruments should be provided that identify the LINQ-aptitudes and unfold them into Java.

Another issue is increasingly affecting not only query languages, but other syntactic constructs like list comprehensions as well, namely *how* to process the produced query results more efficiently. These results can originate from different data sources, not only relational but also hierarchical and OO, and in practice, operate on different memories, main and secondary-memories. Processing of large amounts of data is indeed impractical to be performed in main-memory, and afflicted with mapping constraints when moving from one memory to another. It is unrealizable to move temporary objects to secondary-memories and inadequate to feed main-memories with large data from the secondary one. Typically, the results will be inserted in lists on the object level, eventually producing huge temporary data structures, which should be removed at the end. The procedure is comparable to the idea of materializing the relational results, and moreover similar to the practice of containers in Java, trusting the garbage collector to take care of the rest. Instead of evaluating the query result in its original structure, it can be modified in a lean but more sophisticated way, so that the same effect is culminated without having to process the large results to perform specific operations.

That being said, the question is whether Java will tag along. Functional languages have their theoretical foundations in the lambda calculus, which stimulate the needs of list processing very well. Java and most imperative languages have indeed their advantages elsewhere, but here, they lag behind. The complex problems experienced have often simple roots. For instance, querying complex results is comparable with iterating over lists. This work will put these paradigms into practice and focus upon notable but simple problems. Many querying idioms lack semantic basis. It would be advantageous to retroactively amend the findings with mathematical ties. There are algorithms known in academia concentrating on similar problems that need to be revived.

1.3 Task Description

The topics handled over the course of this thesis are broached in two main passes. First, the possibilities of querying persistent data type-safely and functionally within the JPA framework. Second, the feasibility of processing query results as object populations along with other similarly-constructed ones having different origins.

Both LINQ and JPA will be studied first to spot areas of overlap and to identify tangible gaps. This stage of revision does not steer the thesis in a textbook-direction, but rather aims to reduce the disparity between the two techniques and to contribute for a better understanding before seeking further decisions. The idea of implementing a full-featured translation from LINQ to JPQL will be extensively studied over the course of this thesis, accompanied by a number of general proposals to improve the procedures needed for prototyping for future considerations, avoiding the fallacy of making this research a pure technical handicraft as well.

The second part goes one step further, and tries to answer the questions imposed in the problem description. Namely, how to process the large data resulting from executed queries or gathered by other means. ORM mechanisms shoulder responsibility in persisting data on the RDB, making relational data results available on the object level. Due to restrictions imposed by the different storage architectures, data transfer is either slow or unrealizable. These restrictions will be researched and the behavior of the system will be expressed in *simple* programming terms to identify the *complex* problems more clearly. The concept of processing object results will be narrowed down to the idea of iterating over lists. The imperative way of interpreting loops will be compared to that of functional languages. The role of list comprehensions as powerful syntactic constructs will be emphasized, and their integration into Java will be investigated, keeping in mind the possibility to realize functional queries on the OO data model. Subsequently, implementations in this area will be presented, along with self-implied solutions. List comprehensions are reinforced with mathematical semantic background through the use of the monoid comprehension calculus. Finally, the thesis will be looking briefly at how denotational semantics are supposed to help solving the discussed problems by reference to similar paradigms and with regards to the progress done in research so far.

Despite its shortcomings and for the reasons introduced earlier, Java is deployed as the programming language throughout this thesis. Instead switching to another language, describing the problems in the context of Java might take a turn for the better to confront the development challenges faced by the major programming community, the Java community. RDB are set as the underlying data models to be grounded with practical business applications, and resembling a start for further extensions. Additionally, an application scenario will be introduced and used throughout this thesis to portray the difficulties realistically. Towards the end, it will be shown that the proposed scenario can be extended to work on different data models, such as the hierarchical model.

1.4 Related Work

This section presents a review of the published literature relevant to the work discussed in this thesis:

- Prithiviraj studied the translation procedure from LINQ to JPQL. Meta-models of LINQ and JPQL were designed in terms of the Eclipse Modeling Framework (EMF) and a preliminary prototype was implemented in C#. Not all querying operators were supported, and handling more than one entity in a single query is not provided. Additionally, only dynamic queries are considered in JPQL embeddings, not named queries.

His work lays emphasis on the use of EMF in realizing a technical translation without handling the complications arising after query evaluation [Pri08].

- Garcia and Prithiviraj investigated the possibilities of integrating comprehension queries, by explaining the technical aspects of translating LINQ to JPQL [GP08].
- Wen researched the LINQ internal query translation applied by the .NET compilers. His implementation was able to parse LINQ expression queries, transform them into Standard Query Operators (SQO), and return an Abstract Syntax Tree (AST) that represents the original query in terms of invocations to SQO. The translation algorithm was done in terms of EMF as well [Wen09].
- Fegaras and Maier introduced the monoid comprehension calculus and the monoid algebra. Object Query Language (OQL) queries were rewritten first in the high-level calculus form, normalized to remove extraneous intermediate structures, and finally transformed into a low-level algebra to unnest the remaining data before evaluating on the database [FM00]. An object-oriented database management system (OODBMS) called $\lambda - DB$ was developed to process OQL queries based on the calculus and algebra optimizations [Feg04].
- The work of Pearce and Noble discusses optimization techniques on object queries. Their idea of allowing the system to automatically optimize code in sophisticated ways would reportedly provide a better separation between interface and implementation, not burdening ordinary developers with complex coding mechanisms. A query language was implemented as an extension to Java, providing support for querying collections of objects [WPN06].
- Teubner handled relational-hierarchical mapping mechanisms and data models, best resembled through *Pathfinder*. The compiler first translates XQuery queries into relational query plans through the use of extended relational algebra operators. Afterwards, the plans are evaluated by the database based on the XML documents, returning tables, which themselves are transformed back into an XQuery result sequence. The compiler is integrated into MonetDB [Teu06]. Pathfinder is extended by another enhanced query language *Ferry* and its compilation environment *FerryDeck*. Thereby, programmers can use their scripting languages to access databases. In the background, those languages are translated first into Ferry, which is capable of operating over nested, ordered lists, and then get compiled into SQL:1999 statements with the help of Pathfinder [GMRS09].
- Richta and Toth introduce a formal categorical approach to OO database modeling based on the category theory. They discuss the possibility of applying the category theory as a mathematical formalism to cover not only OO models, but also relational and hierarchical paradigms. A set of proofs has been devised for the hierarchical model [RT08].

1.5 Organization of the Report

The next chapter provides a brief overview of the key concepts of LINQ and JPA. It is necessary to recall the two mechanisms to assert their roles in further application design. An application scenario will be presented and used throughout this thesis. More technical issues, which are of importance from the standpoint of implementing type-safe functional queries using the JPA, will be discussed in the third chapter. The idea of prototyping a LINQ-to-JPQL will be debated, taken into account the progress achieved so far, and more importantly

the new features of the next JPA. The effectiveness of existing solutions will be tested against the requirements.

The fourth chapter considers two object populations, which reside in different memories. They are manipulated in a preconceived scenario, in the same manner as typical programmers would do. Performing join operations on both objects collections without further ado, might relieve potential problems, especially when operating on large data sets. Some of the interim solutions will be presented. In the fifth chapter, list comprehensions will be discussed, especially their contribution in solving similar paradigms in functional languages. The necessity and possibility of implementing comprehensions in Java will be depicted. A simple self-implied code will be performed, which has the Java barriers in contemplation. The last section of chapter five discusses the semantic foundation fostering list comprehensions, applies the findings of research to their practice, and raises the question of integrating denotational semantics generally in programming languages and specifically in the discussed issues.

Finally, the last chapter wraps up with a substantive discussion of the problems faced and solutions proposed throughout this thesis. The take-home message will be given out after outlining a slew of considerations to be drawn for future investigations.

Chapter 2

LINQ and JPA Querying Capabilities

The possibilities of performing type-safe functional queries over OO data models are best initiated by exploring existing techniques aiming to bridge the gap between such models and relational databases. Two of the most arguable query frameworks operating on the OO level and targeting RDB, namely LINQ of .NET and JPA of Java, will be presented. Although this chapter is not intended to provide a comprehensive look at one technology or another, it will hopefully serve as a sufficient overview to the main features of both mechanisms and help as a start before diving into more details in the following parts of this thesis.

2.1 Language INtegrated Query (LINQ)

LINQ is praised by its founders as an efficacious remedy, making use of similarities among the data models operations and trying uniformly to access all kinds of data domains. LINQ is a component of Microsoft's .NET framework for querying data sources like RDBMS or XML files. A detailed exploration of LINQ's principles and strategies will be skipped in this thesis for the sake of brevity. Some distinctive characteristics, that might be relevant for further investigations, will be however explained briefly.

2.1.1 Distinctive Features

In order to facilitate a better understanding of LINQ the following points summarize the key features that worth to be highlighted:

- A LINQ query is constructed either as a *query expression* (aka textual query expression) or as a *method expression* (aka SQO). Query expressions are simpler and provide a number of elementary operations (from, join, select, ...). The operations represent clauses, that are put together declaring a query, and eventually translated *internally* by .NET language compilers into method calls, the SQO. The query expression syntax supports only a subset of the LINQ functionality. The internal **translation from query expression into method expression** is however hidden from the user. The following query expression:

```
IEnumerable<Employees> result =
  from emp in Employees
  where emp.empNo < 10
  select emp;
```

is translated into this equivalent method expression using SQO:

```
IEnumerable<Employees> result =
  Employees.Where(emp => emp.empNo < 10);
```

Wen explained technically how such a translation can be achieved in his work [Wen09].

- The **SQO** are categorized into two kinds depending on which interface they implement `IEnumerable<T>` or `IQueryable<T>`. The arguments that were passed to the methods, extending `IEnumerable<T>` and operating on in-memory collections, are captured by the returned enumerable object. The query results are returned, after employing the logic of the query operator and while enumerating that object. On the other side, methods extending `IQueryable<T>` build an expression tree instead of implementing queries. The queries are represented with these trees. The source `IQueryable<T>` object takes care of handling the query [NET06].

- LINQ offers functional qualities to the .NET languages in many forms. **Lambda expressions** for instance originate from the lambda calculus, most known in functional programming languages. Lambda expressions are used to assign a chunk of code (the anonymous method) to a variable (the delegate) using a parameter list, followed by the => token and by an expression or a statement block. Lambda expressions represent a compact version of anonymous methods to declare the function logic in-line, which can be resolved to both an anonymous delegate and an expression tree, i.e. it is possible to pass the same declarative lambda expression to both IEnumerable and IQueryable extension methods. Explicit return types and explicit parameter types are not needed, leaving this task to be determined by the compiler. By definition, Lambda expressions are anonymous, i.e. do not have identifiers [Kle08].

- Instead of the `IEnumerable<T>`, it is possible to use the type `var`. Based on the results of the query, which is `IEnumerable<T>`, the compiler will automatically infer the type from the assignment (**type inference**), assuming it is a variable of type `IEnumerable<T>`. The compiler performs code error checking and optimization already at compile time, not at runtime. The **type safety** check leverages the querying capabilities offered by LINQ compared to other conventional query languages [MEW08].

- **Expression Trees** represent the essence of LINQ extensibility technique, by enabling the language to be deployed theoretically for any data source. The type of the source collection implements either `IEnumerable<T>` or `IQueryable<T>`. In the first case, the local LINQ query engine is executed; and in the second case, the expression tree-based implementation is invoked. The expression trees are then forwarded to the LINQ providers [PR07].

- **LINQ providers** are implementation of a specific data source that enable the LINQ queries to be used with the data source. They examine first the query trees handed over as expression trees, and finally generate methods at runtime, that are executed when running the query [Kle08]. Not all providers support SQO. More importantly, however, evaluation of the same query is different from provider to provider [GP08]. LINQ to SQL translates LINQ queries into equivalent SQL ones and returns the result as objects, after executing them on RDBMS. LINQ to Entities targets any DBMS having an ADO.NET adapter [Kle08].

- The ability of LINQ to XML and of LINQ to SQL to access XML or SQL data sets lies in the fact that it concerns extensions, which abstract the data as trees and finally apply LINQ operators on them [Kle08]. **Extension methods** are new language features of

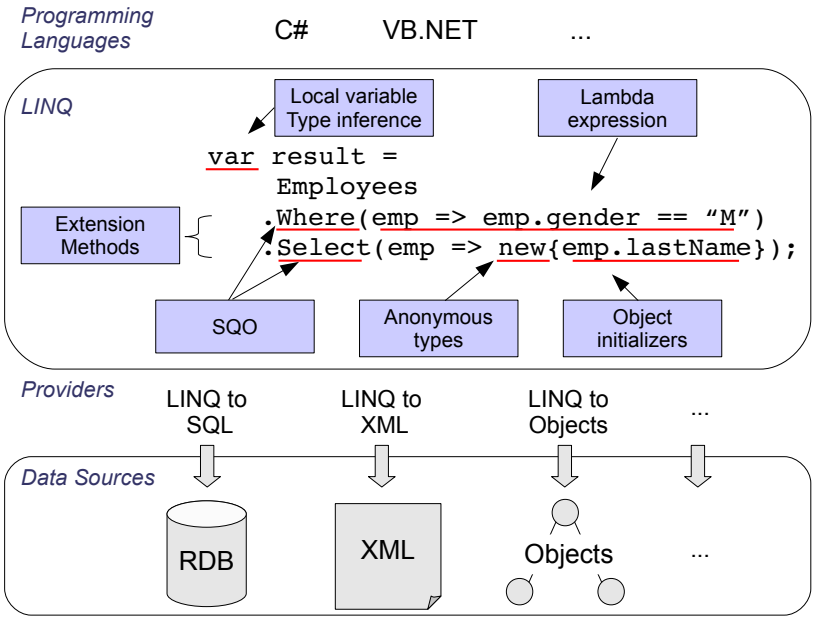

Figure 2.1: LINQ and its environment: The .NET programming languages make use of LINQ to query different data sources through suitable providers. The LINQ query is written with the help of SQO. Here it selects the last names of all male employees.

.NET languages that allow adding methods to existing types. Creating a new derived type, recompiling, or otherwise modifying the original type is priorly not necessary. They provide syntactic sugar for writing queries, which can be optionally deployed by other languages. They are characterized by having the keyword *this* before their first argument. Yet, the declared argument is not passed when consuming the method. They can be called only on instances, not on a class, but still can be defined in a static class [MEW08].

Figure 2.1 illustrates the main features of the language and its environment. ORM can be realized under LINQ in several ways. Depending on the target applications, designers can employ mainly LINQ to SQL or the Entity Framework.

2.1.2 ORM in LINQ

LINQ targets many data sources such as RDBMS. LINQ to SQL and the Entity Framework address ORM techniques in LINQ. A detailed comparison has been listed recently by Sanguanchai[1]. Best practices using LINQ to SQL are manifested by Giesenow[2]. Microsoft is de-emphasizing LINQ to SQL, while putting more effort into the Entity Framework[3]. Not to forget NHibernate, which is the Hibernate ORM version for the .NET languages[4]. A description of ORM from LINQ to a RDBMS will not be further discussed within the scope of this thesis, since it will not be used later.

[1]LINQ to SQL vs. Entity Framework: http://www.osellus.com/blogs/2009/06/04/linq-to-sql-vs-entity-framework/

[2]Why use LINQ To SQL?: http://dotnet.org.za/hiltong/archive/2008/02/01/why-use-linq-to-sql-part-1-performance-considerations.aspx

[3]Is LINQ to SQL dead? http://reddevnews.com/articles/2009/01/01/is-linq-to-sql-dead.aspx

[4]NHibernate: https://www.hibernate.org/343.html

2.1.3 Review

There is no doubt that LINQ plays an important part in changing the way programmers interact with data positively, more so is the fact that it concerns a standard set of operators. The type safety and functionality are the highlights, putting the language in a higher league compared to existing solutions. Some aspects though need to be considered and investigated further to assert their role. The fact that Microsoft is behind LINQ and the fact that LINQ is enjoyable only within the .NET world might encourage some and discourage others.

Auto-completion assistance in the IDE Microsoft Visual Studio helps among others to enforce type safety at compile time [Kle08]. Debugging LINQ lacks more transparency. Many details are hidden from the user level, unlike the situation when debugging normal .NET language constructs. Writing blocks of arbitrarily complex logic in LINQ is possible, but hard to step through. Using the for-each statement, many SQO return elements when iterated over without actually executing any work until the first element is requested. The first element can be suspended until the next element is called. No work is applied internally for any of these statements, until the returned query is iterated over. Controlling the execution of potentially lengthy work is referred to as *Deferred Execution*, which separate errors from their causes. Additionally, *lazy loading* take place when enumerating over results might lead to unexpected results with incautious assignments [MEW08].

Different kinds of auxiliary naming must be used in LINQ when formulating a complex query: iteration variables (emp), lambda expression variables, and structure field labels, which are reduced when expressing queries in SQO notation. Lambda notation like *emp=>emp* add syntactic sugar that can be complicated to explain for average programmers. Other functional programming languages avoid such syntactic overhead accomplishing the same job. As in any other language, understanding efforts are needed to be able to use LINQ safely. In order to fully understand the succinct SQO-syntax queries, the long expression-syntax queries must be first comprehended.

2.2 Java Persistence API (JPA)

JPA is the standard Java persistence specification for ORM, which is part of the EJB specification and J2EE platform, and partially of J2SE. JPA consists of a standard runtime API, a standard mapping definition, and a query language, JPQL. The main concepts behind realizing persistence in JPA are summarized in figure 2.2.

2.2.1 Setting the ORM

The standard persistence descriptor, known as *persistence.xml*, configures persistent support in the entity manager to read and write to a given database. The configuration file can contain one or more persistence units, but each are distinct from each other, are uniquely named, and referenced whenever an application needs to know more about an entity defined in it. Additional attributes within each persistence unit determine, among others, the underlying implementation of the JPA entity manager and vendor specific properties, known as persistence provider [BK06].

Persistence providers, such the proprietary Oracle TopLink[5] and the open source projects Hibernate[6] and EclipseLink[7], also known as ORM persistence frameworks or platforms, implement the JPA specifications, offering developers more services, and the opportunity to access data in RDB within a persistence framework. Even if no corresponding features

[5] Oracle TopLink: http://www.oracle.com/technology/products/ias/toplink/index.html
[6] Hibernate: https://www.hibernate.org/
[7] EclipseLink: http://www.eclipse.org/eclipselink/

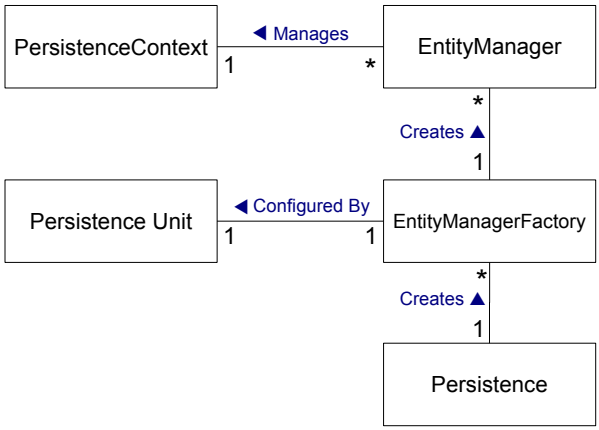

Figure 2.2: An in-depth look at concepts of JPA: Entities are considered regular non-persistent Java objects until an *entity manager* is utilized to manipulate them, either by direct reference or when reading from database. As of now, those persistent objects are entities run by entity managers, which on their turn are obtained from *entity manager factories*. The *persistent unit* configures the entity manage factory and the *persistence context* of an entity manager contains all managed entity instances. One persistence context can be referenced by more than one entity manager. Java instances with the same persistent identity are not allowed within a specific persistence context [KS06].

exists in JPA, access to such RDBMS is configurable with the provider's specific mappings. The vendor is provided through the persistence descriptor with information on the database dialect (MySQL, Oracle, etc.), JDBC driver, its address and database used, login, caching, as well as other settings. Mapping of JPA objects takes place through the package `javax.persistence.annotations` and according to settings in JPA persistence.xml and orm.xml. Providers extend the JPA standard runtime API by additional packages and annotations, and additionally manage association between classes, e.g. one-to-many and many-to-many relationships [BK06]. A sample snippet showing how to set up the persistence file using Hibernate as a persistence provider and MySQL as a RDBMS can be found in the appendix A.2.

In the sample persistence.xml, a used ID and a password for accessing the target database was set. However, in practice, the database is defined as a data source at the application server, along with the login information. The data source will be then named, and referenced in the Java application whenever the database is accessed. This alternative way spares from defining a configuration file for each utilized environment, and is more transparent. The code in the Java application is capable then of accessing the database using the same data source name in all environments. Each environment could have its own user ID and password for the database and it would be defined at the data source level [MD05].

employees is a sample database acquired from Launch Pad[8]. The database shapes a typical cliche used often in the database research, ideally for testing purposes. The database is of a large imaginary corporation having tens of thousands of data concerning its employees, departments, department-employees, department-mangers, titles, and salaries. Figure 2.3 shows its structure, which was taken from mysql.com[9]. The six tables have a total of about four million records. The database was first imported into the local MySQL, and is manipulated as needed to meet the demands of this work.

The ORM can be defined alternatively in code as annotation and discovered automatically

[8]Sample database with test suite: https://launchpad.net/test-db/

[9]The employees sample database on mysql.com: http://dev.mysql.com/doc/employee/en/employee.html

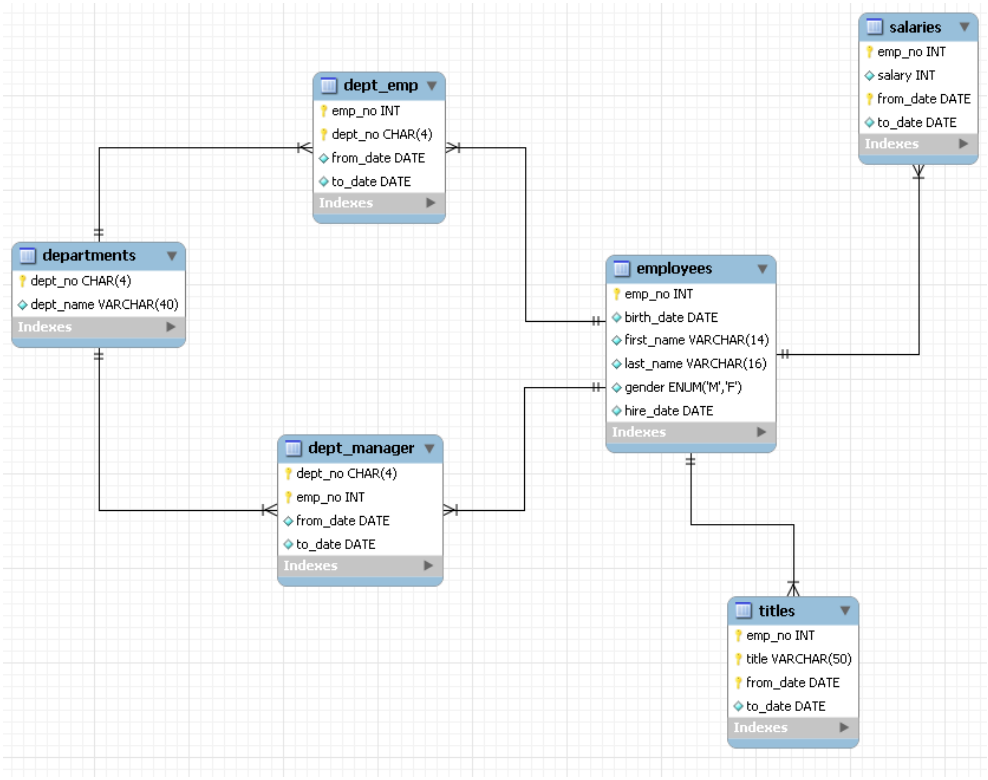

Figure 2.3: The employees sample database: The ca. 160MB-big database is a typical cliche used often to describe large imaginary databases, ideal for testing purposes. The database will be frequently mentioned throughout this report, and all example code will always refer to it.

by the persistence class management. In either way, after synchronizing content between the relational data structure and the OO structure, tables from the database like employees, departments, etc. will be mapped into the entities Employees, Departments, etc. , and the table attributes such as title (of an employee) or name (of a department) will be mapped into member variables of those entities, along with getter and setter methods to manipulate the variables. An ORM of the table employees into entity Employees would yield a similar code to the following:

```
@Entity
@Table(name = "employees", catalog = "employees")
public class Employees implements java.io.Serializable {
private int empNo;
private String lastName;
...
@Id
@Column(name = "emp_no", unique = true, nullable = false)
public int getEmpNo() {
  return this.empNo;}
public void setEmpNo(int empNo) {
    this.empNo = empNo;}

@Column(name = "last_name", nullable = false, length = 16)
```

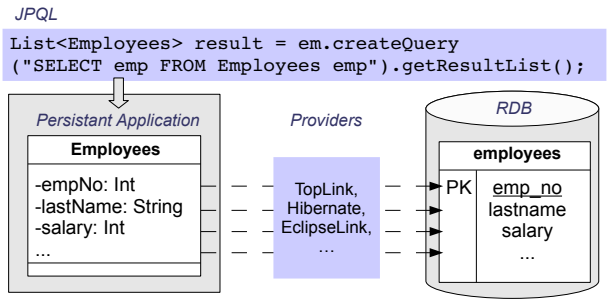

Figure 2.4: Persistent object view of an existing RDB table: JPQL queries an entity (Employees), which its persistent mapping to an existing RDB table (employees) is managed by a JPA persistence provider. The mapping definitions are stated in the XML descriptor or as annotations in the mapped entity itself.

```
public String getLastName() {
  return this.lastName;}
public void setLastName(String lastName) {
  this.lastName = lastName;}
...
}
```

Information about the relationships with other tables is also written down in the mapped entity. In the sample database schema, employees has a one-to-many relation with the table dept_emp (department employees table), which has two columns one for employees number and one for department number. The mapped entity is accordingly processed. The data set here is fetched lazily when it is first accessed:

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "employees")
public Set<DeptEmp> getDeptEmps() {
  return this.deptEmps;
}
```

The mapping techniques differ from a persistence provider to another. For instance, the mapping of the same previous table within the mentioned schema resulted in two slightly different entities when using EclipseLink and Hibernate. Hibernate provides detailed annotations in the mapped entity. Determining the method body for those classes that are constructed from the ORM mapping is generally a challenging matter, and is handled also differently from provider to another. Nevertheless, an entity is just one of three different types of persistable classes that can be used with the JPA and should be supported by the providers. The other two are mapped super classes and embeddable classes, which all represent managed classes, that are made persistent by a provider [KS06].

Within the Eclipse IDE, a framework called *Dali JPA Tools* is utilized to supply definitions and tools for the editing of ORM for JPA entities and to perform reverse engineering. The framework is characterized through automated wizards and programming dynamic assistance to help programmers to reduce the complexity of mapping[10] .

2.2.2 JPQL in Action

JPQL comes into action as a standard querying tool that enable Java applications to interact with entity instances. It defines operators and expressions based on the used abstract per-

[10]Dali JPA tools on Eclipse: http://www.eclipse.org/webtools/dali/

sistence schemes of entities and their relationships, and performs searches against persistent entities independent of the relational back-end store mechanism. The persistent entities are loaded, modified, persisted, deleted and queried by simple API, as soon as the mapping is defined. Figure 2.4 portrays the use of JPQL to query entities, in order to fetch data from relational tables. A JPQL query boils down to persistent fields of the entity or its association to another entity. The result of querying a persistent field within an entity will be, at the worst, zero or objects of the same type as the field itself [KS06]. Most of the SQL keywords are adopted one-on-one, and many obtain under JPQL additional functionalities.

JPQL Queries can be set either as strings or embedded as a Domain Specific Language (DSL) in code. A string query is parsed by a query engine into a syntax tree before interpreting the ORM for each entity in every expression, and finally translated to an equivalent SQL expression. Specifying queries as strings can be error prone, making queries weakly typed while operating against strongly-typed Java objects. The IDE needs further validation, auto-completion and refactoring to overcome such problems. The subsequent code segment shows JPQL queries set as strings:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory
    ("FPGA");
EntityManager em = emf.createEntityManager();
List<Employees> employeesList =
    em.createQuery("SELECT emp FROM Employees emp
                    WHERE emp.empNo < 10").getResultList();
// Processing the result
em.close();
```

Entity manager factory and entity manager were first initialized, pointing at the persistence unit named FPGA. Afterwards, the JPQL query was executed and its result was assigned to a Java list of type Employees. The project configuration details along with a number of code examples are listed in the appendix A.1 and A.2. Embedding JPQL as type-safe internal DSL has the advantage of checking errors already at compile time, and reviewing type agreement between operands to the contrary of specifying the query as a string. Prithiviraj studied IDE Customization of JPQL and showed in detail the benefits of embedding JPQL as an internal DSL. Besides, JPQL meta-model and the accompanying grammar based on JPA 2.0 specification were developed and well-formedness rules were additionally implemented to ensure correctness of the embedded query that are not covered by the grammar [Pri08].

2.2.3 JPA 2.0 and Criteria API

The development of JPA 1.0 is the result of the JSR 220 Expert Group that defined it as part of the EJB 3.0 specification, which is itself part of the Java EE 5 platform. The proposed final draft of JSR 317 regarding JPA 2.0 was published in March 2009 [Mic09]. Starting from Java EE 6, JPA is no longer part of EJB. The API gained in importance since the last version and attained first-class vendor support, especially of the JBoss community[11].

JPA 2.0 provides support for both optimistic and pessimistic locking[12]. ORM improvements, such as modeling of collections, maps and lists, not representing entity relationships are supported through `@ElementCollection` annotations. Also, mapping of unidirectional one-to-many relationships is possible, to the contrary to JPA 1.0, where only bidirectional one-to-many relationships were allowed. JPQL is amended to have SQL-like capabilities[13].

[11] JBoss: http://www.jboss.org/

[12] Multiple instances of the same entity can be active in the same time in optimistic locking. In pessimistic locking, the entity is locked in the database for the whole time while being in the application memory [Möl09].

[13] Article discussing the new features of JPA 2.0: http://www.sdtimes.com/content/article.aspx?ArticleID=31655

The highlight of the JPA 2.0 is a type-safe API based on a meta-model of the managed classes in the persistence unit. Advantages of the meta-model API include IDE auto-completion support, refactoring, compile time and protection against SQL injection. The type-safe *Criteria API* supports everything that JPQL can do. Queries are created by calling methods of Java objects. The main goal of it is to avoid runtime exceptions thrown while parsing JPQL, and to allow constructing of complex dynamic queries. The API is similar to the Hibernate Criteria API[14], enabling users to write both string (as JPQL) and embedded queries.

To use it, first a meta-model is generated out of the annotated entity (of Employees of section 2.2.1), and constructed in a similar manner to:

```
import javax.persistence.metamodel.*;
@TypesafeMetamodel
public class Employees_ {
  public static volatile Attribute<Employees, Integer> empNo;
  public static volatile Attribute<Employees, String> lastName;
  ...
}
```

Afterwards, queries can be formed in the following way:

```
// Defining JPA persistence entity manager
CriteriaQuery<Employees> qEmployees = qb.create(Employees.class);
Root<Employees> employee = qEmployees.from(Employees.class);
q.select(employee.get(Employees_.lastName));
```

Other operators than select can be attached, such as selectDistinct, and, equal, like, greaterThan, join, and others. Type safety is guaranteed since lastName will return the type stated in the meta-model class, i.e. String. Gavin King, founder of the Hibernate project, is trying to add additional features to impose type safety all the way through to the query result set[15]. The above-stated expression would be extended to include:

```
...
Query<Employees> eq = em.createQuery(qEmployees);
List<Employees> res= eq.getTypedResultList();
// Processing the result
```

CriteriaQuery returns a Query, instead of directly the results. In this way, other things can be done via the Query interface, such as parameter binding. Query will have Employees as type parameter, if it will return `List<Employees>` type-safely. Compared to the LINQ version above, the syntax is indeed more verbose, especially when sub-queries grow. Related to the verbosity, Java Persistence 2.0 specification lead Linda DeMichiel commented: "*We did make trade-offs in the meta-model API between the granularity of types and the complexity of the API*"[16]. Further on, the main features of the proposed Criteria API and its technical differences to JPQL are listed on the same blog. An UML class diagram of the meta-model criteria packages can be found on eclipse.org[17]. The Criteria API scales up the JPA certainly to a higher level, type safety tasks in JPQL are broken for the time being. Even without the proposals of King, JPA is still cutting the right corner.

The new specifications and especially that of the Criteria API are still not fully developed at the moment of writing this report. EclipseLink is implementing JPA 2.0 currently[18], which

[14] Hibernate Criteria API Documentation:https://www.hibernate.org/hib_docs/v3/api/org/hibernate/Criteria.html

[15] Gavin King discussing type safety after query execution in the Criteria API: http://blog.hibernate.org/Bloggers/LindaBlogsTheTypesafeQueryAPIForJPA20

[16] Official blog of Linda DeMichiel on Java Persistence 2.0 proposed final draft: http://blogs.sun.com/ldemichiel/entry/java_persistence_2_0_proposed

[17] eclipse.org: http://wiki.eclipse.org/images/b/b1/Uml_class_diagram_metamodel_criteria_packages.gif

[18] EclipseLink JPA 2.0 implementation status: http://wiki.eclipse.org/EclipseLink/Development/JPA_2.0

will be supported by GlassFish as an application server[19]. EclipseLink plans to make its implementation compliant to any JDBC, including Oracle, MySQL, PostgreSQL et al. , along with other performance-related features[20]. Additionally, OpenJPA has set a road map for implementing JPA 2.0[21].

2.3 Summary

Over the course of this section, key terms of LINQ and JPA were briefly explained. Such a revision is vital to have an overview of familiar solutions before going further steps, and setting blueprints from the scratch for wheel reinvention. It is necessary to know first where the two mechanisms intersect, and where do they distinct from each other. Not everything described will be immediately advantageous, but at least the light has been spotted and kept for later reference.

Three aspects of LINQ are of particular importance for the present research. First, type safety is automated through the compiler and IDE compile time support and not left to the user as in JPQL. Second, LINQ is equipped with an armada of providers making it possible to query almost any data source. A small set of them are supported directly by Microsoft though. JPA addresses solely relational back-ends. Finally, .NET languages are notably leveraged by the functional properties of LINQ. It is possible now to effectively and succinctly iterate over collections, unlike the typical way known formerly in C# or VB.NET.

Innovations to support type-safe and functional querying are not fully supported or still missing in JPA. Whilst JPQL queries operate over strongly-typed Java objects, they themselves are weakly typed. Even when embedding JPQL queries in code, constraints imposed by the grammar have to be studied to construct syntactically correct queries. JPA 2.0 is enriched with type-safe querying capabilities through the new Criteria API. In any case, Java programmers are left alone to master iterating over query results, probably ending up by utilizing the old good for or while loops.

Information on the used tools and vendor specific settings were outlined and others were skipped, just to stay focused on the main issues. The thesis will take advantage of the sample database and deploy it for testing purposes. These matters seem secondary at first sight, but in reality a portion of time was invested to put such necessary steps on track. The next chapter will discuss the technical burdens to realize type-safe functional queries. Delving into the practices and nuances of implementing such an automation might boost ideas collected so far or decrease the pace.

[19] As announced by Linda DeMichiel, Java Persistence 2.0 Specification Lead, November 2008: https://slx.sun.com/files/LindaDeMichiel-JPA2.pdf

[20] As announced by Gordon Yorke, an EclipseLink Architecture Committee member, November 2008: https://slx.sun.com/files/GordonYorke-EclipseLink.pdf

[21] OpenJPA development process for JPA 2.0: http://openjpa.apache.org/jpa-20-roadmap.html

Chapter 3

Type-safe Query Processing

LINQ and JPA technologies have benefits of their own and are designed for specific use cases. They both tend to narrow the gap between OO abstraction and the underlying models in a standardized way. In this chapter, ways to implement type-safe functional queries on Java will be discussed. A closer look will be taken upon the design aspects that are needed generally to fulfill such expectations. The need of compile time checks of the queries will be outlined. LINQ-to-JPQL translation will be brought up for discussion, taking in mind the new features in JPA 2.0. Existing implementations relevant to the strived design aspects will be demonstrated as well.

3.1 Design Aspects

The proposed final draft of JSR 317 published in March 2009 adds new features to the current specifications of JPA, such as a meta-model API, support for validation, and the type-safe Criteria API as indicated in the last chapter 2.2.3. However, the modifications will not affect JPQL drastically. It will stay roughly the same for the next couple of years. Instead, more effort has been done for improving querying manners with the referred Criteria API [Mic09]. It has been discussed how LINQ is benefiting from specific qualities, and the question now is, whether it is possible to delegate some of the eminent features to JPA/JPQL to be eventually deployed on Java? A translation from LINQ to JPQL would enable developers working on the Java platform to perpetuate their work without being forced to switch to another environment. Or is it worth to play a lone hand, and deploy LINQ-like capabilities in Java using JPA?

Related work has been achieved paving the way for a possible translating of LINQ into JPQL, as stated in the introduction 1.4. Adding LINQ-like capabilities using JPA has been commenced by a number of applications as well. More will be said about these experiences in later parts of this chapter. Both scenarios though need first to obey certain rules, primarily compile time checks and other related descriptions.

3.1.1 Compile Time Checks

Queries can be integrated as strings or embedded in code. Strings are simpler but erroneous, since detecting misspellings in the query syntax or the names of entities is possible only at runtime. The task of writing syntactically correct queries cannot be left for users, since typically users tend to write faulty queries, mostly because they do not fully understand them and interested rather in their results. Embedding the whole query as a type-safe embedded DSL enables the detection of errors already at compile time. IDEs like Eclipse or Microsoft Visual Studio provide syntax checking and code auto-completion, each for the languages it supports. As soon as the code is saved, it will be compiled in the background and checked for syntax

errors. Such a help can be irreplaceable when thinking about embedding queries in code, for example in Java. Syntactically correct queries do not automatically mean well-formed ones though, best practiced in [Pri08]. In the same work, embedding JPQL in Java is advocated, and well-formedness conditions for the JPQL embedded-queries are listed. Further on, such conditions that are not captured by the JSR 317-based BNF grammar were pinpointed, and necessary steps to implement the well-formedness conditions and to check well-formedness during editing were depicted.

That being said, it is important to discuss how error checking should be done. Error reporting can be delayed and be first chronicled by the compiler when running faulty queries, but this would not work properly in general. Simply because there is no direct access to the source code, and practically hard to pursue such errors. Getting error messages at some level of an intermediate language is inadvisable [Par07]. They should be collected at the source code level instead, for example by building ASTs within the translation design that are able to check out against error conditions at an early stage, earlier than starting to process the queries. If a faulty query is transformed embedded as a functional call or a message call, and subsequently applied by the compiler to check for further conditions, error messages from the compiler will pop up with respect to the intermediate code. Yet, truth be told, those errors reported are hard and sometimes impossible to understand.

It is clear by now that errors should be determined already at compile time and not first at runtime. Therefore, examining the validity of the queries based on syntactic and well-formedness conditions should be carried out at compile time. Additionally, if errors are reported, then conditions should be rated. The error messages can be channeled and then presented as an error string, so that the programmer can inspect these errors [AP07]. The application could be told about the error object, error description object, and relate to the line in the source code, and then once these objects are supplied to the application, corresponding information could be displayed. Semantic conditions and predicates can be verified, when placing the check within the input grammar that throws an exception upon failure [Par07].

However, this is one side of the story. The other side requires further tracking. Type checking is needed in both ways, on the way to evaluate a query against a data set, and when results are fetched on the way back. Taken together, type checking means walking over the entire AST, verifying that typing rules are fulfilled at each tree node, and eventually computing type of the expression represented by the node. Suppositionally, syntactically correct and well-formed queries are run against a data set. Since the evaluation process is executed and returned to the OO level, the result of the query must be by definition some object, but the type of the object is not easy to determine automatically. Type checking of the returned objects is required for further processing. Using unparameterized objects is indeed against all safety principles. If the type is known beforehand, then the locally-defined object of the same type can be assigned to the query result object, and they should coincide without fail.

```
Iterator <Departments> iterator = // Query result of type Departments
// Further processing
```

One solution would be to keep a record of the type associated with each object. Whenever the object is referenced in the application, this type is checked against a symbol table, aka identification table. A look at the AST is necessary to be sure that the result types are actually the ones expected [AP07]. There are queries that have different types, and for those there should be a way of ensuring that the types of expressions are consistent. Moreover, checks should be executed to verify that the declared object is assigned a type that exists. As mentioned earlier 2.1.1, LINQ overcomes this problem by imposing type inference using `var`. The compiler then decides dynamically the type of the object when it is evaluated. In the newly introduced Criteria API, type safety is handled differently. Considering:

```
CriteriaQuery q = qb.create();
```

```
Root<Employees> emp = q.from(Employees.class);
Join<Employees,Departments> dep = emp.join(Employees_.departments);
```

The argument of the join method here is of type `Set<Employees,Departments>`. It is not possible to construct an invalid join from the customer root, because the parameterization of the join method guarantees type safety. King's proposals referred in 2.2.3, discuss type safety after execution queries, which is an important matter not explicitly formulated in the current JSR final draft.

Generally ASTs should be built in a way, that they are able to examine syntax checking, well-formedness, and type checking conditions themselves, not depending on the default compiler support to accomplish this task. Upon colliding with the first violation of checking conditions, the AST should proceed its work, not only reporting the first error messages, but also including multiple error messages, which can emerge later. For this reason, reset points should be defined, to make sense of parsing of the remaining token [AP07]. Further below, the mechanical steps needed for the proposed translation will be pinpointed, as human-understandable text as possible, but first maintaining functional properties will be discussed.

3.1.2 Functional Qualities

A LINQ-to-JPQL translation will not forward the functional qualities of LINQ to JPQL. Lambda expressions for instance cannot be deployed per se on Java. They introduce new syntax, similar to that of functional languages. LINQ favors a declarative approach in iterating over results [MEW08], in contrast to the imperative approach deployed in Java and most imperative languages. LINQ's effective iteration technique cannot be simply delegated to JPQL as well, nor to Java itself, for reasons that will be explored in detail in section 5.1.3. Executing *functional* queries on the Java platform is fraught with complexity related to the imperative nature of the language, the same as in C# and other imperative .NET languages. The remaining paragraphs of this chapter will focus on performing type-safe queries on Java. The upcoming chapters shoulder the burden of supporting functional characteristics generally in Java, which can be forwarded to query processing.

3.1.3 Mechanical Notes

Primarily the compiler performs some operations on the received input query, manipulates it, and finally sends it to be evaluated. The operations are applied on the AST, and include syntax and well-formedness checking, type checking, flow analysis, code optimizations and others. The coming lines review implementation steps based mostly on type checking, as an example of possible examination. ASTs are the most practical way to translate input structure into code. In contrast to conventional parse trees, ASTs possess only related nodes with the input symbols. Parse trees are bigger, because they contain nodes for all rules required for input recognition, and hence delicate against any changes to the parser grammar [Par07].

Different kinds of nodes are defined in the AST. Query clauses like `select`, `where`, etc. can be defined as operator nodes, identifiers like `TableName`, `ColName`, etc. can be defined as identifier nodes, and the corresponding classes will be produced `OperatorNode` and `IdentifierNode`. Upon parsing and matching patterns in a query, the AST gets enlarged by adding a level, which represents a new node that is being instantiated to be the root, and previously built subtrees are made children of the new tree, which is returned (and thus may in turn become a child). Visitor patterns are used to easily navigate through ASTs, mainly to avoid changing the generated code of the AST by hand, whenever new functionalities are defined over the AST [AP07].

Supposing that the compiler is asked to perform type checking of a query expression. A visitor object, say `TypeCheckVisitor`, is first constructed by the compiler, and subsequently the `Accept` operation on the AST having that `TypeCheckVisitor` as an argument

will be called. `Accept` is implemented by tree nodes by calling on the visitor. Operator nodes would then call the operation `VisitOperator` and identifiers would call the operation `VisitIdentifier`. The `TypeCheck` operation that used to be in class `OperatorNode` is now the `VisitOperator` operation on `TypeCheckVisitor` [1]. States can be accumulated by the visitors, as they visit each element in the object structure, e.g. upon finding the first match, visitors continue their work. Other operations can be implemented in the same manner. Taken the work done by Wen [Wen09] into consideration and provided that JPQL is understood, writing the visitor from LINQ using the SQO to JPQL should be theoretically not difficult. Such a visitor might help in evaluating a LINQ read-only query on a main-memory object population. Additionally, it can also be useful to do the same on an ORM-mapped object population, e.g. by forwarding JPQL queries to the ORM engine. More on main-memory and secondary-memory object scenarios will be discussed in the next chapter.

In static-typed languages like Java, every expression E is either ill-typed or has a static type, which can be computed directly without evaluating E[2]. This can be particularly advantageous, namely enabling static type checking since evaluation of such expressions of static type T will always return type T. Thanks to the bottom up algorithm over the AST followed in most static-typed languages, many types of expression AST leaves will be identified at once. Furthermore, the type of the children nodes and the type rule for that kind of expression infer the types of internal nodes [AP07].

IDE specific frameworks can be used to facilitate the translation process, such as the EMF[3]. In this case, a meta-model of the parsed tree aka concrete syntax tree is needed in Eclipse, which should be built according to the grammar defined for the parsing process. All implementation steps have to act out according to the meta-model. Type safety will be enhanced thereby. Actually, the same way the Criteria API tend to do in the near future. Whilst EMF is indeed a powerful tool that can be used during prototyping of such a translation, especially when utilizing the Eclipse IDE, it surely brings along further aspects that should be taken into consideration. It is believed that the use of EMF meta-modeling will go beyond the scope of the discussion handled in this thesis, which tries not to swamp the reader with technology-related details. An effort on describing the required tools should be set. Instead, a conscious decision has been made to invest more time in managing the whole process rather than communicating one or two technology-specific views. The know-how experienced by the works of Prithiviraj [Pri08] and Wen [Wen09], and more generally Garcia [Gar08] can be used excellently to deploy EMF in relation to query processing.

3.1.4 More on Translating LINQ to JPQL

Many language specifications do not define a standard meta-model of their languages. Besides, there are no fixed rules to come up with an appropriate meta-model neither. Generally, a look behind the grammar curtain of such languages is needed in order to understand them fully. In the case discussed here, related language meta-models, like the Object Constraint Language (OCL), can help constructing a meta-model for JPQL, one which covers all the operators than that of [Pri08]. [Gar08] discussed among others, meta-modeling and transformation algorithms of OCL from a model-driven software-engineering point of view. Figure 3.1 visualizes the general road map that can be taken for translating LINQ queries into JPQL ones. As mentioned a moment ago (3.1.2), LINQ-to-JPQL translation will not delegate the functional features of LINQ to JPQL.

Considering the work done already in this area, a meta-model for LINQ's textual syntax has been developed, and the translation done into SQO uses only a subset of that. Since one

[1] More details and code examples on type checking using visitor pattern: http://www.lasagnej.org/Lasagne_J_Visitor_DP.htm

[2] Ill-typed expressions are those ones that are incorrect at compile time, but turn out to be valid first at runtime.

[3] Eclipse Modeling Framework Project (EMF): http://www.eclipse.org/modeling/emf/

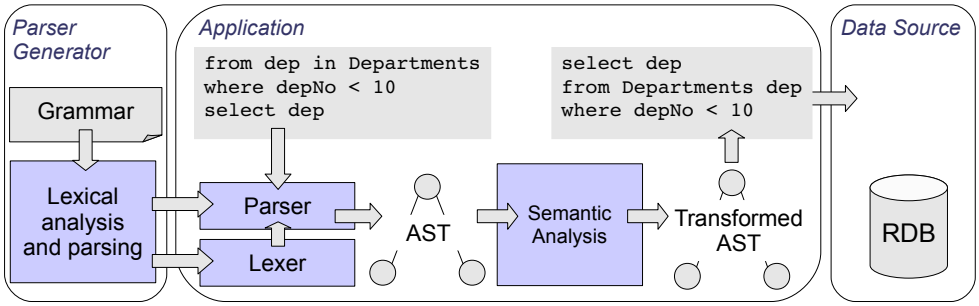

Figure 3.1: Query translation: Lexical analysis and parsing tools generate lexer and parser according to the grammar file defining the expected syntax of the input query, which is represented as an AST. Several operations are applied on the AST of the input query before transforming into the desired output query. Semantic analysis include type checking. Finally, the query is sent for evaluation. Here, a LINQ query is translated to an equivalent one in JPQL.

of the goals is to enhance the main-memory evaluation of such queries, a LINQ expression consisting only of *pure* SQO would help accomplishing that. The expression would have only the irreplaceable building blocks.

LINQ expression syntax:

```
from d in departments orderby d.Employees.Name,
  d.Total descending select d
```

LINQ SQO syntax:

```
(departments.OrderBy(d=>(d.Employees.Name))
  .ThenByDescending(d=>(d.Total)))
```

More functional syntax:

```
OrderBy( departments, d=>((d.Employees).Name),
  d=>(d.Total))
```

The idea of rewriting queries before sending them for evaluation was already mentioned in the introduction 1.2. The rewriting might not improve the performance of the system when dealing with one or two queries, but surely when handling tens or hundreds of them. The next section describes briefly the impact of the new JPA on the LINQ-to-JPQL translation.

3.1.5 More on JPA 2.0

Web applications often provide their users with complex search forms. JPQL in its current nature will be executed twice in the background to manage the situation. JPQL query will be created in the first round, and parameters will be binded in the second one. The Criteria API accomplishes the whole procedure once, avoiding redundant and erroneous code forwarding, and making it more dynamic. The most striking benefit in the new API, however, queries will be parsed by the Java compiler, without the need for extra syntactic validation, auto-completion and refactoring support. As shown earlier in 2.2.3, annotations are a fundamental component in the newly proposed API. However, on-the-fly annotation processing are not fully supported by most IDEs. Features like this should be taken care of, once the proposed API is finalized.

The prime motive behind a possible LINQ-to-JPQL translation is the work done and knowledge gathered so far. Nevertheless, the main restraint is the emerge of Criteria API.

LINQ-to-JPQL translation is bounded with respect to implementing the mechanical steps discussed above, which are time consuming. Moreover, it is questionable whether a translation of this kind would be worthwhile. Before getting back to prototyping work, the next section gives an insight of the main LINQ-like implementations so far.

3.2 Language Implementations

There a number of solutions out there, assuming to be a replacement for LINQ on the Java platform. The next paragraphs present two major ones, *Quaere* and *JaQue*, which strive type-safe and functional queries, inspired by the advantages of LINQ.

3.2.1 Quaere

Quaere consists of a query DSL and two engines for running queries: Quaere for Objects and the incomplete Quaere for JPA[4]. In-memory object graphs are run on the object query engine. However, its performance deteriorates when run against large number of objects[5], since it deploys non-indexed in-memory object graphs [6]. Evaluation of the queries can be done in a single traversal of an object collection. The following snippet selects the department called "Sales":

```
// Defining JPA persistence entity manager
Iterable<Departments> qDepartments =
  from("d").in(entityManager.entity(Departments.class)).
  where(eq("d.getDeptName()","Sales")).
  select("d");
// Followed by a loop to iterate over qDepartments
```

Superficially seen, Quaere is similar to LINQ. A more precise look refutes this hypothesis. Quaere interpreted type safety rather narrowly as the current implementation does not provide full compile time checks. Embedding the entity methods (`d.getDeptName()`) as strings is error prone, as it happened when trying to execute the simple example above after several runs. The query qDepartment is not evaluated lazily, as in LINQ. That means memory processing will take place while fetching the data and while iterating over them. The JPA engine is still being developed, promising enhancements. There is no further documentation about the internal structure of the implementation.

3.2.2 JaQue

Java integrated Query (JaQue) is primarily not a DSL, but rather a library that provide LINQ-like capabilities on Java, including type safety, language integration, expression trees and other features [Tri09]. The open source API makes use of ASM for building expression trees. ASM is an open source middleware, which is capable of modifying available classes or dynamic generation of classes, straightforward in binary form[7]. The frequently maintained framework generates byte-code from non-Java source code, and in JaQue's case LINQ source code. JaQue fully implements LINQ's expression trees and use them to build queries in a DSL. The promised features can be put in perspective by considering JaQue as a possible competitor to LINQ. JaQue makes use of *closures*. The following lines will describe closures in general, talk about the structure of JaQue and conclude with focusing on its JPA support.

[4] Quaere project: http://quaere.codehaus.org/

[5] Tests made against large data sets using Terracotta as persistent store: http://tusharkhairnar.blogspot.com/2009/04/querying-java-objects-stored-in.html

[6] As stated by one of the project's developers: http://markmail.org/message/q4trcvhxalyjkjlf/

[7] ASM: http://asm.ow2.org/

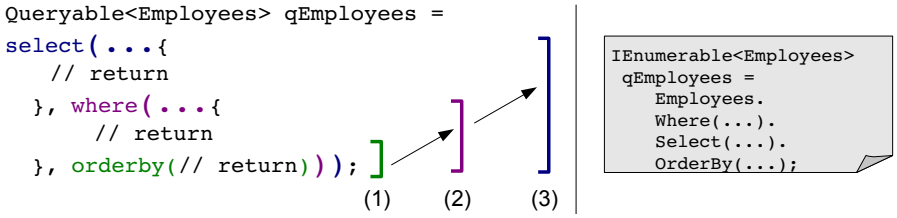

Figure 3.2: Formulating queries as closures: A closure is a way of passing code from one object to another. This concept can be adopted in querying collections. Here, the select clause passes code to the where clause, which in its turn hands over to the next clause. The outermost object finally executes the code. If the query was something like *SELECT emp FROM Employees emp WHERE ... ORDER BY ...*, then the *ORDER BY* subset is executed first (1) and its result is given back to *WHERE*, which makes the same (2) and returns the final filtered result to *SELECT* (3). The LINQ code on the right side is just for comparison. The example here is simplified, but in reality more verbose, containing a number of background classes and interfaces. It would have been clearer if closures were integrated properly into Java.

Closures

Closures are anonymous functions that *close* over their surrounding scope. They posses different names and syntax over a number of languages. In Java, they have the following syntax: { *formal parameters* => *statements expression* }. The function, specified by the closure, is executed and therewith getting access to all the local variables that were in scope when it was created. In other words, closure is a way of passing code from one object to another, offering the possibility to the other object to execute the code. For instance, the function `{ int x => x + 2 }` takes a single `int` argument `x` and returns its value incremented by 2. Using an `invoke` method, a closure can be invoked: `{ int x => x + 2 }.invoke(10)` will call the closure with argument 10[8]. The integration of closures into the next planned Java 7 has run into difficulties though, and will be for the time being not part of it[9]. The API is also operational without the use of closures, but more verbose however as illustrated in figure 3.2.

Structure

Similar to LINQ, JaQue has providers, which ports the library into different target structures, such as Objects, XML, and JPA, and is intended to be fully operational with other JVM languages, such as Scala and Groovy.

`Query`, `Expression` and `Operation` are the major classes that characterize JaQue. They can be used with static import. `Query` contains a big number of methods, that mainly deal with object streams and prepare them for further processing, such as forwarding the `where` clause to the `select` clause in `select(where(...),...)`. Beside the `Query` class, a set of unchangeable rules are defined in the `Expression` class to be applied. The rules can once instantiated, deployed everywhere, and can be extended to form other rules, such as `or(greaterThan(...),lessThan(...))`. Finally, the `Operation` class supply the library with a number of operations for manipulating object streams, such as aggregation operations [Tri09].

LINQ's lambda expressions are covered by the use of closures, which entail verbosity. Internally, the library builds ASTs from lambda expressions using ASM byte-code analysis.

[8] Java Closures: http://www.javac.info/

[9] Blog of Mark Reinhold, head of the development department of Java Standard Edition at Sun Microsystems: http://weblogs.java.net/blog/mreinhold/

Assuming that *E* represents an expression, *L* a lambda expression and *P* the AST building function, then *P(L(E)) = AST(E)*. In terms of normalization, immutability, cacheability and other properties of AST, some issues are considered and being further researched. The LINQ expression trees are completely implemented, and constructed using a parsing method and ASM, unlike the .NET way of letting the compiler build the trees. Function are taken as parameters for the tree parsing method [Tri09].

JPA Support

Unlike LINQ expression trees, the JPA interface is not fully implemented in JaQue. Simple projection, filtering and aggregation operations can be still performed though, as the following example shows [Tri09]:

```
// Defining JPA persistence entity manager
// Defining the JaQue entity manager
Queryable<Employees> qEmployees =
  select(new Function<Employees, Employees>() {
      public Employees invoke(Employees e) throws Throwable {
      return e;
    }
// Further closures (where, order by, ...)
},jem.from(Employees.class));
// Followed by a loop to iterate over qEmployees
```

Where `jem` is the name of defined JaQue entity manager. Despite the use of closures and the contiguity to LINQ's methodology, the query is indeed clouded with code segments, which will soon propagate as the query grows.

3.3 Summary

This chapter discussed design aspects that have to be taken into account when considering LINQ-to-JPQL query translation or a LINQ-like query processing using JPA. In either case, prototyping needs to be aware of compile time checks and some technical restrictions on the one hand. On the other hand, emulating functional qualities of LINQ cannot be achieved with a magic wand. LINQ offers the .NET languages functional opportunities, whereas a LINQ-to-JPQL translation does not mean also functional qualities automatically on Java. The mechanical steps described the technical procedures that are required to realize type-safe queries in general, and were reviewed abstractly keeping in mind not to bore the reader with detailed technology-specific how-tos. Some available implementations were presented at the end, addressing the same problems and roughly coinciding with the thesis's objectives.

Up to this point, a crossroad has been reached and another piece of the jigsaw towards accomplishing this thesis has to be set. Whether to put the gained knowledge into practice and concentrate on realizing a prototype that translates LINQ to JPQL, or rather go on to manage a wider view, intertwined with issues of deep relevance that have yet to be manifested later on. The first option would mean full-featured type-safe implementation on the JPA framework, without exploring the functional capabilities of LINQ. The prototype should be built upon the progress done so far by Prithiviraj [Pri08] and Wen [Wen09], and should take into consideration the shortcomings of existing solutions like JaQue [Tri09]. After assessing the advantages and disadvantages and pointing out the necessity of such pure technical handicraft, a conscious decision has been made to prioritize the second option. Therefore, JaQue will be utilized in the upcoming chapters for demonstrating purposes instead of a self-implemented solution.

Other arguments fostering the second approach, are the pledges undertook recently by the proposed draft of JPA 2.0. The idea of translating LINQ to JPQL can be of interest for academia, and should be theoretically not hard to realize. Nevertheless, there will be two main concerns. First, the functional characteristics of LINQ will not be forwarded to JPQL. Second, the emerge of the new Criteria API in JPA 2.0 has indeed a negative impact on the future of JPQL. The simple string-based JPQL queries are likely to be replaced by more sophisticated and type-safe ones. It is only a matter of time, until the Java community can enjoy the same privileges, granted by LINQ to the .NET world. JPA 2.0 is heading the right way. It is believed that a translation of this kind would be out-of-date and would arouse little attention. Research should focus on the new features of JPA 2.0 or of LINQ without trying to convert from one highly-featured language to another one with uncertain future, and ultimately without proselytizing for a particular branch. Along with other important insights, adding functional finishing touches to the achieved type-safe queries will be particularized in the next chapters.

Chapter 4

Manipulating Object Populations

A result set is produced after executing type-safe queries, retrieved from the relational level and represented as objects in an application. Other objects operating on the main-memory are to be iterated along with the query result to perform certain operations, typically joins. This chapter shows restrictions of iteration mechanisms and reveals potential problems. Followed by interim solutions that might leverage the inefficiency when scanning large sets of data, taking into consideration the constraints that encounter manipulating objects located in different memories or even in different data sources. The behavior of the system will be illustrated through a number of examples based on a preconceived scenario to devote key characteristics for further investigations.

4.1 Processing of Query Result

The application scenario to be discussed in this section and the following ones, makes use of the sample database and its ORM settings mentioned in 2.2.1.

4.1.1 Iteration over Result Objects

Suppositionally, the corporation reports severe loss because of the recent credit crisis, and wants to lay off some of its employees. The CEO decides to close one of its departments, writing red figures. Eventually all the employees working in this department will be integrated into other ones or become candidates for a possible downsizing. The CEO does not need to retrieve the data concerning the departments and their managers extra all the way from the RDB. The departments list is all written down in an application for fast reference, since it does not change as often and not big as the other tables. The manager list contains twenty four records regarding the nine departments of the corporation, the old and current managers. It has a total of four columns, department number, employee number, and from to period, to state the period of management of this employee. The list can be loaded technically as objects of the class DeptManager on the object-level, e.g. :

```
DeptManager m1 = new DeptManager("d006","499928","2002-07-30,
                "9999-01-01");
```

Based on a class structure which is constructed as an entity by reverse-engineering means from the RDB. The year 9999 indicates that this employee is the current manager of the department. By considering the department *Quality Management* as the black sheep, the CEO wants a list of all employees of this department (with number *d006*) along with further information of their manager. By using JaQue's JPA implementation as exemplified in 3.2.2,

it is possible to fetch all the needed records from the dept_emp table (department employee) to the mapped entity DeptEmp as objects, using an ORM framework. The table contains employee number, department number and the period to state the employment time in the related department. The query would look like:

```
Queryable<DeptEmp> qDeptEmp =
  select(new Function<DeptEmp, DeptEmp>() {
  public DeptEmp invoke(DeptEmp t) throws Throwable {
    return t;
  }}, where(new Predicate<DeptEmp>() {
    public Boolean invoke(DeptEmp s) throws Throwable {
      return s.getDepartments().getDeptNo().equals("d006");
    }}, jem.from(DeptEmp.class)));
```

Deploying the object model of the application environment, the persistence implementations enable loading of objects from the secondary-memory as active main-memory objects, and take care of performance-related issues, as caching to avoid unnecessary round-trips to the RDBMS. The mapped DeptEmp contains about 320,000 entries. An employee has one-to-many relation to both DeptEmp and to DeptManager, because employees change their positions over time. Aquiring all DeptEmp entries without further filtering is theoretically possible. However, it might cause the application to run out of memory. Even after increasing the heap size for the runtime, ORM is generally not suitable for transferring large data sets into main-memory [BK06]. Instead, if the filtering condition is known beforehand, then refining the result is more efficient when performed by the RDBMS. For the sake of experimenting, a try to fetch all those salaries (ca. 3 million entries) shed the light on a recent bug in Hibernate, not being able to deal with scrollable large data sets[1]. The workaround for the moment would be to avoid scrollable results and use pagination instead.

Before going on, some notations are redefined in the course of this chapter and the following one to avoid ambiguity. A *collection* is defined as a data structure that represents a group of objects, such as a list, a stack, a tree or a query result. Yet not to be confused with Java Collections. An *iterator* represents a design pattern to walk over all elements of a collection. Iterators generally do not guarantee a specific order in which the elements are accessed [ZHR+06]. Though, some ORM providers return the query results sorted by default, which can be configured not do so. After fetching the desired data from the RDB level to the OO level as persistent objects. These objects are then walked over for further processing. A simple iterator over the object result can be constructed in Java using the for or alternatively the while loop in the following form:

```
for (Iterator<DeptEmp> itrDeptEmp=qDeptEmp.iterator();
      itrDeptEmp.hasNext();){
       DeptEmp e =(DeptEmp)itrDeptEmp.next();
       // Do something
}
```

qDeptEmp is the result of the query being executed as shown above. As of Java 5, it is possible to use the enhanced for-each statement, which is similar to the C# for-each used to iterate over a LINQ query:

```
for (DeptEmp e : qDeptEmp){
  // Do something
}
```

[1] Hibernate bug reports: http://opensource.atlassian.com/projects/hibernate/browse/HHH-3576

The for-each statement preserves all of the type safety, while removing the declaration of the iterator and its generic idiom. It is not usable for filtering, when elements in a list need to be replaced, or for iterating over multiple collections in parallel. The benefit of using the first iterator is that it can be used for manipulating the collection (e.g. delete items) [GJSB05]. Within the scope of this thesis, the for-each will be utilized instead of the conventional for or while loop, and whenever an iterator is mentioned, generally both the iterator in the above for loop or the for-each statement can be meant. Thanks to the ORM mechanism, it is possible through the entity DeptEmp to access the methods of other entities referenced (e.g. `e.getEmployees().getLastName()`), the same way the tables are associated with each other on the relational level. If the operation should print numbers and names of the employees working in the mentioned department and regardless of the loop used, the result would yield over 20,000 entries according to the sample database:

```
10009: Sumant Peac
10010: Duangkaew Piveteau
...
499963: Danny Lenart
499964: Randy Matzov
```

Due to the above-mentioned heap memory restriction, a filtering condition was put in the query. Anyways, the runtime difference for filtering the query beforehand (on the relational level) is much faster than doing it later, e.g. by putting an if condition in the loop (on the object level):

```
// qDeptEmp fetches all entries in dept_emp
for (DeptEmp e : qDeptEmp){
  if (demp.getDeptNo().equals("d006"))
  // Do something
}
```

Up till now, two collections are first brought in, one containing all the managers, and one having all the employees associated with these departments. Subsequently, they will be sought to determine the unfortunate people according to their department number or name. In addition, the manager's name will be captured. Technically, two loops will be performed over collections that originate from different storages, primary and secondary one, also known as *main* and *secondary-memories* respectively. The outer loop contains objects on the main-memory, whereas the inner loop iterates over objects that originate from the result of the query of the secondary-memory. Finally a join is executed, say printing names of employees and their managers. If the condition is satisfied, then specific methods (e.g. print names) can be called:

```
for each department manager in collection DeptManager in main-memory do
  for each department employee in collection DeptEmp in second-memory do
    if (department manager, department employee)
      Print list
```

Similar code in Java:

```
for (DeptManager m: qDeptManager){
  if (m.getDepartments().getDeptNo().equals("d006")){
    for (DeptEmp e : qDeptEmp){
      if (m.getDepartments().getDeptNo()==
          e.getDepartments().getDeptNo()){
          // Print list
      }}}}
```

Whereas qDeptManager is the locally-produced object population. If the called method shows in addition the name of the manager of this employee, then the result would be similar to:

```
10009: Sumant Peac, Onuegbe
10010: Duangkaew Piveteau, Onuegbe
...
499963: Danny Lenart, Pesch
499964: Randy Matzov, Pesch
```

Different managers pop up for different employees, even though they all work in the same department, and have one manager. Because the sought department had four different managers over time, the query result was joined for all those four managers, i.e. the original result was given out one time for each manager. Further optimizations can be implied to the query by adjusting the filtering conditions in the query itself. Additional conditions can be added on the application level to help specifying the current manager, e.g. only the current managers having the data attribute set to `todate` ("9999-01-01"):

```
for (DeptManager m: qDeptManager){
  if (m.getToDate().equals(todate))
  ...
}
```

This time the correct result is displayed. The CEO is now capable of deciding who should stay and who should leave the corporation. The examples above assume that only a relational back-end store is deployed, which is mapped to the object level by ORM means. The next section explores possibilities of extending the scenario to possess additionally a hierarchical data source, utilizing the same tools.

4.1.2 Hierarchical Structures

Java Architecture for XML Binding (JAXB) enables Java objects to be marshaled into XML and the inverse, XML to be unmarshaled back into objects. The back-end data have then hierarchical structures or trees. Trees are organized in linked nodes and can be seen as a particular case of graphs. Each node of a tree has a unique parent (e.g. employees) and either zero or several children (e.g. employees, salaries, departments) [2]. Persistence providers such as Hibernate[3] and EclipseLink[4] can be extended to work with persistent XML data, objects from a RDB are marshaled into objects in an XML document, and the other way around. The library JaQue provides also support for XML. So theoretically, the handled scenario over the course of this thesis can be adjusted to include operating on different data structures, relational and hierarchical, covered by persistence mechanisms of Hibernate or EclipseLink. Teubner mentioned in his thesis ways of translating XQuery into a special form of relational algebra operators before evaluation on a RDB [Teu06].

Iterating over objects originating from hierarchical structures would be principally identical to iterate over RDB objects, since the providers take care of the mapping implications and represent them both as Java objects. Considering a hierarchical back-end store, full iteration is comparable with traversing a tree, i.e. visiting all tree nodes with regard to their orders and hierarchy. Without using object-hierarchical mapping strategies, iteration is achieved normally by the use of tree iterators, which are built a bit different, making use of the tree structure of XML data. They process nodes and select only those that pass filtering conditions applied to one of the parent nodes. Depending on the condition, the entire tree branch

[2] JAXB Reference Implementation Project: https://jaxb.dev.java.net/

[3] Relational persistence for XML objects with JAXB and Hibernate: https://www.hibernate.org/218.html

[4] Setting EclipseLink's JAXB: http://wiki.eclipse.org/EclipseLink/Examples/MOXy/JAXB

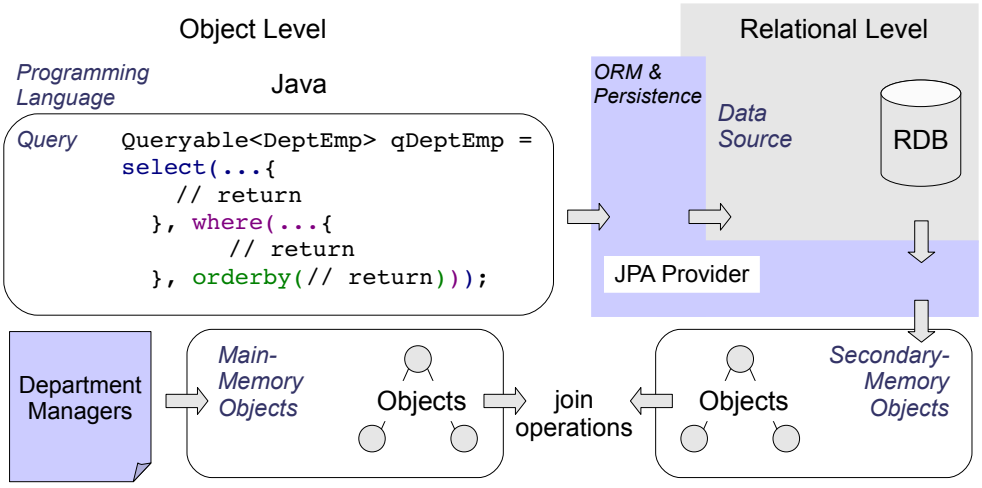

Figure 4.1: Operations between objects residing in different memories: The discussed use case handles two object populations, one is produced as a result of a query, which is persisted by an ORM framework. The other one is constructed locally. The main-memory object population (here DeptManager) can also originate from other models, e.g. relational or hierarchical, but all operate on the object level.

referring to this node will be processed or not. Otherwise, children of this node should be ignored. In a linked list, population is proportional to the length of the list. In a tree, population explodes exponentially in the height (depth) of the tree [Wei98].

4.2 Objects Residing in Different Memories

The scenario described until this moment represents a general situation that encounters many applications. Data are brought into the object model, having different modeling backgrounds, and might be residing in different memories (shown in figure 4.1). They are all represented as objects, and typically saved into lists for further manipulation by an OO programming language.

4.2.1 Secondary-memory Objects

In this scenario, the objects that are located in the secondary-memory come from a RDBMS. They are subject to ORM vendor-specific configurations, and denoted in this work as *secondary - memory objects*, although they are actually active main-memory objects, at least for a while. The ORM providers manage caching, transactions, sessions between the object and the relational level, and enable using such objects as if they are any other Java objects. The persistence mechanism of such providers is responsible for getting them at some agreed time back to the secondary-memory. The session and transaction implications are different from one provider to another. Providers tend to hide the complexity behind the mapping from the user-level [BK06]. It is possible though to create *pure* main-memory objects out of those secondary-memory ones, e.g. to set the state of the session in Hibernate to *transient*, making the main-memory objects not related to any data on disk[5]. Additionally, using Oracle as a RDBMS, Java classes can be loaded into the database, producing pure disk-based objects. Cases like this, are suitable when performing heavy SQL calls on the Oracle database server

[5]Interface session on Hibernate: https://www.hibernate.org/hib_docs/v3/api/org/hibernate/Session.html

itself. The Java classes can do periodic SQL queries and generate reports (files) on a file server. By loading the Java classes in the Oracle database server, performance improvements should be expected in comparison to be run outside the database server[6]. Yet, data transport has to be taken into consideration.

Views of queried data can be materialized[7], first implemented by Oracle RDBMS[8]. In relation to the scenario discussed, materialized views can be used to optimize fetching data from the database, i.e the secondary-memory objects.

4.2.2 Method Calls on ORM-mapped Objects

Through the use of code instrumentation reported in the last chapter 2.1.1, LINQ can obtain an OO view on different kinds of data sources. On the other hand, mapping technique of JPA, demonstrated in 2.2.1, focus on getting an OO view on the RDB only. One data source is adopted for simplicity reasons in the course of this thesis, rather than handling more complex situations, like realizing a common view of a RDB table and an XML representation.

Hibernate and co. mainly define setter and getter methods to access member variables in the entity, as exemplified in the entity Employees 2.2.1. They offer their users the possibility to manipulate mapped entities, by adding their own methods and variables (e.g. `setZodiac(String)`, and `getZodiac()` of an Employee). After synchronizing content between the object and relational level, those changes will not get lost. However, that might cause some confusion. Supposing that a join operation is performed between two object population having different sources, say main-memory Employees and secondary-memory Employees. The main ones are constructed within the application, whereas the secondary ones are fetched by ORM means from a RDB. Now, when applying a method defined in the Entity, e.g. `getEmpNo()`, then there should be no problem on both populations, all employees' numbers will be returned. Yet, when `getZodiac()` is asked, then only those main-memory Employees will answer back, but for the secondary-memory Employees there will be no data to fetch, since they are not stored from the first place in the RDB, and there is no corresponding zodiac column. Specifying the bodies of such methods needs further investigation, to avoid trying to acquire data that does not exist on the relational level.

4.2.3 Memory-related Restrictions

The simple algorithm that joins two collections, here DeptManager and DeptEmp, handles a *nested loop join*. Each object from one collection (the outer entity) is compared with each object from the other collection (the inner entity) looking for objects that satisfy the join predicate. The total number of objects compared and, thus, the cost of this algorithm is proportional to the size of the outer collection multiplied by the size of the inner collection. The cost grows quickly as the size of the input collections increases. In practice, the cost will be minimized by reducing the number of inner objects that are considered for each outer object. Moreover, the outer collection should be smaller than the inner collection [YM98], as in the case discussed. The use of nested loop joins makes only sense when joining small subsets of data, and if the join condition is an efficient way of accessing the second collection [CB04]. Otherwise, the performance will degrade notably as witnessed in this work. Moreover, the produced code is *bad*, in the broader sense that it is inefficient and gets verbose as collections and predicates grow.

More importantly, these queries are being executed and displayed regarding collections residing in different memories. As a matter of fact, this problem unleashes another similar

[6]Loading and dropping Java objects into Oracle: http://www.oracle-training.cc/teas_elite_util9.htm

[7]Materialized views are pre-computer (materialized) query results. They outrun external views by persisting their results to the database, and thereby support faster access to pre-computed information [CB04].

[8]Oracle Materialized Views & Query Rewrite: http://www.oracle.com/technology/products/bi/db/10g/pdf/

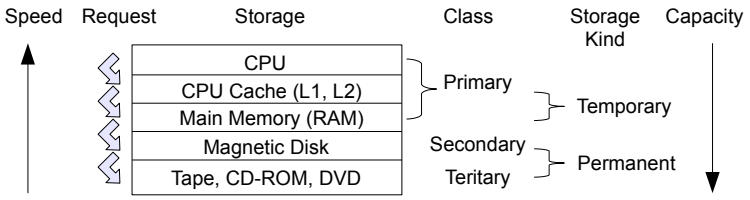

Figure 4.2: Memory hierarchy: Data are requested from lower levels in memory hierarchy, and processed in higher levels. Size of address space in main-memory is limited and may not be sufficient to map large databases. Additionally its cost is much higher than secondary or tertiary storages. The later ones are nonvolatile, suitable of making DBMS data persistent across shutdowns or crashes but still slower than primary storages [YM98].

issue that was referred in 4.1.2: Not only querying simultaneously over data sets located in different memories, but also in different structures, e.g. relational and hierarchical data sources, like LINQ is reportedly capable of, and eventually building a join operation. The problem appears at the surface mostly, when looping over large results to find a matching condition and then execute the desired operation.

Main-memory objects are volatile, kept temporarily but are up-to-date and faster to retrieve. To the contrary of the secondary-memory objects, they are persisted to disks, kept permanently but slower to get access to, depending on the ORM strategies. Large databases do not usually fit in main-memory, and keeping the result objects there is inflexible. Besides, storing in the main-memory has to take the limited storage capacity into consideration, as opposed to the secondary-memory. Apart from the expensive search that result in nested join loops, query processing is brute force, and neither concurrency control nor reliability in case of a crash is guaranteed [YM98]. Figure 4.2 reviews the main characteristics of the different memory kinds.

There are storage architectures other than the two-level main and secondary-memory architecture, like the one-level and the third-level architectures. The first one is also known as main-memory database system. As the name says, concerns only operations on the main-memory and is suitable for applications needing fast access. This architecture gained importance due to the decrease of cost, the availability of non-volatile (battery backup) main-memory, and the increase of main-memory sizes[9]. The later architecture possesses a tertiary storage, such as optical discs or tape drivers, and is indeed the future model for applications with enormous sets of data, which cannot be handled sufficiently by a secondary storage [YM98]. In the meanwhile, the two-level storage architecture is likely to remain typical for most database applications.

When data items are fetched from the secondary-memory by specifying their address, the cost depends generally on the order in which the items are accessed. This is due to the fact that disks are normally divided into blocks, the far the items are located from each other, the more expensive is the process of fetching them, as pictured in figure 4.3. Modern architecture models try to minimize access costs by imposing caching techniques [CB04], and in the case of persisted data, ORM mechanisms apply their caching policies. Moreover, most RDBMS allocate some space, buffer, to keep data fetched from secondary-memory into main-memory for some time before being pushed away by other data. The system is relieved from performing additional disk operations when future requests for items are located in the same block as the current data. There are different methods that manage discarding unused blocks. The ORM providers optimize data access and transfer between main-memory and secondary-memory [BK06].

[9]The Asilomar Report on Database Research: http://www.sigmod.org/record/issues/9812/asilomar.html

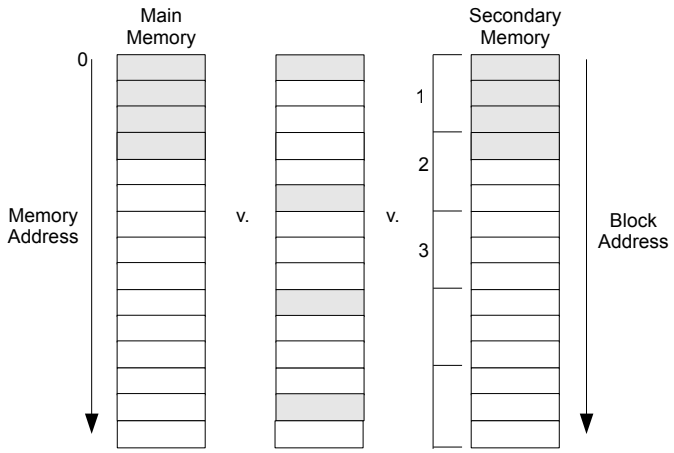

Figure 4.3: Data access by comparison: Time of database operations is dominated by the time for I/O operations. The costs of accessing four neighboring items in main-memory is the same as accessing four items resided farther apart (as in the middle structure). Unlike secondary-memory, which is divided into blocks forming the unit access, here two block operations are needed to bring in the four items [CB04].

It has been discussed that fetched queries should be optimally manipulated already on the relational level, before being allocated to objects on the application level. There will be cases, however, in which particular requirements are changed. That means, data should be re-transported again. When handling small data sets, there will be no problem with that, quite contrary to the large ones. Suppositionally, the heap memory problem from above is enlarged to be able to fetch one third of the selected data set of the salaries table (e.g. only those salaries that amount *more* than 50,000 Euros), then this would mean having one million objects. There will be indeed difficulties impeding transporting this large collection from one memory to the other one, whenever a change in the predicate is realized (e.g. those salaries *less* than 50,000 Euros), or whenever a small update takes place (e.g. deduct 50 cents for postal charges from every salary). Transporting large data is a fundamental problem in computer science, and in the case discussed, large query results and other object collections, which are located in different memories. The case handled within the scope of this thesis is simplified for clarity by considering one RDB as a data source. Furthermore, the situation described so far handles a join between two collections. In real-life scenarios, applications can deal with greater number of data sets undergoing several conditions.

4.3 Operations on Objects

ORM techniques help to manage data access and transfer [BK06]. Apart from the data transport issue, there will be objects in main-memory in practical scenarios, and for those it would not be easy to compute the join predicates. The size of the collections to be iterated plays an important role. Search of an in-memory collection would be affordable if the collection's size is small and would be easier if an index to the target attribute or column is provided beforehand [OO00]. The example with DeptManager handles a small collection of main-memory objects just to demonstrate the case. Computing larger collections would change the situation, say iterate over the employees (300,000 entires) instead of the department managers (24 entries). Assuming that employees are in main-memory, and only those relevant DeptEmp

records are fetched from secondary-memory, the join predicate is computed after a considerable amount of time on a standard machine, which is unbearable for real-world scenarios. The CEO would have got rid first of the database expert who is taking much time for querying those collections. The department employees query can be further refined to match only those that come in question. The manager or employees list cannot be filtered beforehand. Such iterations are ominous, the system's performance is indeed affected negatively.

One of the main operations that can be applied on object results are selection, projection, and join, whereby join is the most expensive one among them [YM98]. Selection and projection will not be mentioned in this section. Instead, ways of improving the work of repeated sequential scans will be explored, especially processing joins between objects located on different memories.

4.3.1 Methods for Computing Joins

The main strategies for implementing join operations are: block nested loop join, hash join, sort-merge join, and indexed nested loop join. A generalization of the simple nested loops algorithm is the *block nested loop join*, which makes use of additional memory to lower the number of times that the DeptManager (M) collection runs. M is scanned once for every collection of M objects. The advantage of block nested loops join is that no indices are required beforehand. Accordingly, they can be used with any kind of join condition [YM98]. Considering B_m and B_e as the blocks of department managers and department employees respectively, a pseudo code of the block nested loop join can be written as follows:

```
for(DeptManager B_m : qDeptManager)
  for (DeptEmp B_e : qDeptEmp)
     for (DeptManager m : B_d)
       for (DeptEmp e : B_e)
         if (m, e)
           // Do something
```

Hash joins might be a solution to handle large data sets. Not all objects will be compared with each other. Instead, comparisons will be applied only to those objects having the same hash code. A hash table is built out of the smaller collection (M) on the join key in main-memory. The larger collection DeptEmp (E) will be scanned to probe the hash table to find the joined objects. Hash joins are applicable if the smaller collection can be stored in available memory. A single read pass over the data for the two collections reduces the cost [CB04]. As join operations get complexer and maybe more efficient, there is a doubt that typical programmers would spend time in developing the needed code to go over those elements in the main-memory. For the sake of demonstrating, the subsequent code shows a possible implementation in Java [WPN06]:

```
HashMap<String,List<DeptManager>> m_tmp =
  new HashMap<String, List<DeptManager>>();
for(DeptManager m : qDeptManager) {
  List<DeptManager> ml = m_tmp.get(m.getDepartments().getDeptNo());
  if(ml == null) {
    ml = new ArrayList<DeptManager>();
    m_tmp.put(m.getDepartments().getDeptNo(),ml);
  }
  ml.add(m);
}

for(DeptEmp e : qDeptEmp) {
```

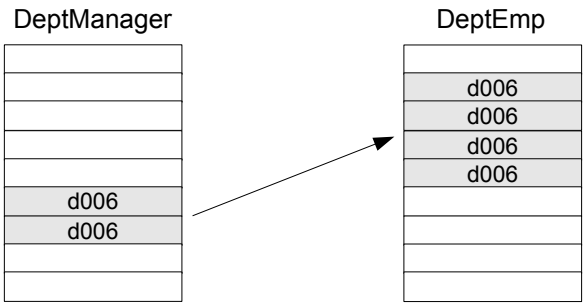

Figure 4.4: Grouping as a solution for nested loops: Instead of iterating over all objects, more convenient would be grouping them according to a specific matching criteria to form smaller groups marked in gray, which can be then iterated and compared with each other object from the same group faster. All other non relevant objects will not be considered.

```
  List<DeptManager> ml = m_tmp.get(e.getDepartments().getDeptNo());
  if(ml != null) {
    for(DeptManager m : ml) {
      if (m.getToDate().equals(todate)) {
        // Print names
}}}}
```

The first code snippet creates a map from Department numbers (as Java String) to department manager objects. Afterwards, a loop over all department employees is performed, and only those having the same department number are considered. Whether to map from department number to DeptManager or to DeptEmp, depends on the size of the collection, and upon that, mapping to the smaller collection is executed. Altogether, the hash-join implementation produces the same result as the aforementioned nested loop join but more efficient [WPN06]. Yet, truth be told, a regular programmer is unlikely to apply this optimization in practice. Even though the complexity is perhaps underrated, but taking in the benefits of the enhanced loop is not seen by those programmer at first sight. *Sort-merge joins* deliver better performance than nested loop when handling large data sets but still are not as good as hash joins in general. They outrun hash joins when no sorting is required, i.e. when the join predicates are already sorted. Each of the involved relations is accessed only once [CB04]. An implementation of the sort-merge join and other join methods are kept aside for the sake of brevity.

Another solution would be forming smaller groups in each collection, and *then* start the iteration. Both DeptManager and DeptEmp are grouped according to the sought department number respectively. There will be a smaller part to compute the joins and subsequently to associate the groups. Only those department managers, and only those employees having a department number d006 are considered, otherwise not. This simple optimization technique, known also as *indexed nested loop join*, saves the intensive redundant iteration over objects that are anyways out of the question. If the condition is fulfilled, then the group is entered and looped, otherwise skipped to the next group. An index over the real elements that qualify has been constructed. An *auto grouping* can be built at compile time in the application to spare the programmer from investing time in writing simple but expensive loops. The drawback of this approach is the need of updating the index as soon as changes has been done in the groups. Additionally, performance is not improved extraordinarily. The next code snippet demonstrates the use of indexed loop join:

```
for (DeptManager m: qDeptManager) {
  if (m.getToDate().equals(todate)&&
      m.getDepartments().getDeptNo().equals("d006")) {
        // Add objects to qDeptManager_d006
}}}

for (DeptManager m: qDeptManager_d006) {
  ...
}
```

The basic forms are further on extended to perform better results. Examples are: Grace hash join, hybrid hash join, and others. Further information can be found in [CB04].

4.4 Summary

This chapter articulates a concern that arises *after* evaluating a query. Two kinds of object collections are formed, the first one operates from the main-memory and the second one is disk-based. The latter objects are loaded as active main-memory objects and kept there for a while, before persisting them back on the second-memory. ORM providers, like Hibernate and EclipseLink, take care of caching and performance-related issues, and hide many complex mapping details from the user-level. Characteristics of both mentioned memories were reviewed, to explain the origins of the problem. Optimizations can be arranged for the secondary-memory objects, like manipulating the query to fetch only the results needed, before assigning values and get iterated. However, moving objects from secondary-memory to main-memory is slow and sometimes precarious, and unrealizable the other way around. Main-memory structures might posses methods and values that have no opposite in the mapped relational database. Further light should be shed upon handling methods of the same class structure, which are defined to operate on an object population, but not recognized by the other one. There will be cases, where data has to be transported, e.g. predicate adjustments or updates. By definition, mixing objects and manipulating them without further ado is not feasible, either the data is unnecessarily large or the predicates do not match in the first place.

An application scenario was set up to demonstrate performing simple collective operations on the two object population. The secondary-based objects can be prepared in advance before being assigned to lists to be iterated, but the main-memory objects cannot. In a double loop, iterating over main-memory objects is not a neat solution, especially when it concerns large data sets. Hash joins, indexed nested loop join, and other joins are optimization idioms to the naive nested loop join. The last section applied join operations on the discussed scenario to underline the problems associated generally when manipulating object populations, the same way a standard programmer would do.

The goal of this chapter was to raise typical problems, like data transport and double processing, which are well known in the database research. They coincide perfectly into the handled use cases, and their solution might help to yield a result, that satisfy the preliminary findings researched in the previous chapters. Computation of the join predicate should be carried out more effectively, and above all legibly for standard programmers. It would be useful if some frequently occurring-patterns are recognized in advance, before entering a loop. The knowledge gained from this chapter, encourages considering other alternatives, like using functional mechanisms inspired by other languages.

Chapter 5

Functional Computing with List Comprehensions

Nested loops have plain code that can be understood and deployed by any programmer, but are inefficient when iterating over large data collections. Other enhanced loops are more complex having little advantage. These views are worthy to be taken seriously for real world scenarios, placing special emphasis on relieving expensive main-memory operations, avoiding data transport and offering standard programmers legible and succinct code. Improvements to entail transforming the simple, inefficient iteration mechanism into a more complex succinct form will be discussed in this chapter. One way to follow is represented by the use of *list comprehensions*.

5.1 Functional Nested Loops

Java and many other prominent languages adopt a straightforward imperative form to iterate over loops, which is also called the statement form. Functional programming languages realize iterating over loops in an additional form known as the functional form, or the expression form [Pol09]. Imperative-language programmers would probably assume that no alternative way can make the computation of nested loops any simpler. Yet, the for or while loop in Java is limited to iterate over a single collection, and can easily get ambiguous as the loop grows. In contrast to the functional languages, loops can be far more complex, and above all more concise thanks to list comprehensions and other higher-order functions. Therewith, nested loops can be specified, the unneeded data skipped, and only those relevant ones will be selected and processed [Alt07]. The definitions stated previously for collections and iterators in 4.1.1 are reused in this chapter.

5.1.1 Expressing Loops as List Comprehensions

List comprehensions have different syntax over a number of languages, the most eminent one comes from Haskell: [$e \mid q$], where q is a qualifier, either a generator in the form of *variable* $\leftarrow$ *sequence expression* or a filter, and e is an expression to be executed. The generators refer to iterators and filters to boolean conditions [Hut07]. For example, the expression [x*$x \mid x \leftarrow$ [1,2,3], even(x)] will evaluate to 2*2=4, since 2 is the only even number in the input list. Suppositionally, the following loop structure is set:

```
for (x1 : Collection1)
  for (x2 : Collection2)
    ...
```

```
for (x_n : Collection_n)
  if p(x_1, x_2, ..., x_n)
      e(x_1, x_2, ..., x_n)
```

The loops define items that iterate over collections, and after passing the predicate, an expression is executed in terms of the items x_1, x_2, ..., x_n, e.g. `join(employee,department manager)`. The if conditions can be indeed rearranged and set after each loop separately. The query can be rewritten from loops into list comprehensions [Bil02, GS96]:

```
[ e(x_1, x_2, ..., x_n) | x_1 ← Collection_1,
  x_2 ← Collection_2, ...
  x_n ← Collection_n,
  p(x_1, x_2, ..., x_n)]
```

The for loops generate the lists (generators), which are then filtered by the predicate (filter), and hence the expression is executed. Both, the predicate and the expression are normally dependent on the elements x_1, x_2, ..., x_n. More precisely, x_1, x_2, ..., x_n are ranging over $Collection_1$, $Collection_2$, ..., $Collection_n$, i.e. x_1, x_2, ..., $x_n \in Collection_1$, $Collection_2$, ..., $Collection_n$ [Kfo08]. Although they seem at first sight unfamiliar, list comprehensions provide succinct syntax and embody a mathematician way of expressing queries. Their functionality will be most appreciated, as the number of loops and predicates increase, and therewith the number of lists generated and the possible understanding effort. Their code can be called a *good* code, in comparison to normal loops. In relation to the scenario described, up to three different data collections will be required to calculate the highest salary of an employee having the title "Graduate Engineer", along with the verbose and the resource-consuming code. .NET languages deploy LINQ as a tool for using comprehensions, whereas comprehensions are not incorporated standardly into Java, or used by any standard extension of it. Learning from experience gained till now, there is a need indeed to find ways of optimizing the code to avoid nested loops, and eventually make use of list comprehensions in some notation. The comprehensions would help in evaluating the object collections returned more efficiently, especially those located in different memories. The subsequent section describes the relationship between list comprehensions and higher-order functions.

5.1.2 Higher-order Functions and List Comprehensions

Higher-order functions are used in functional languages such as Haskell, Python, Common Lisp and Closure. Mathematicians call them functionals or operators, best exemplified by the derivative in differential calculus. A higher-order function can take functions as arguments, and produce others as an output [Hal09]. For developers, they represent a way of writing short definitions for list processing that were previously coded explicitly using simple loops or even comprehensions. *Map*, *filter*, *fold* are examples of such functions. Given that e is a function of x, and p is the predicate to be met in terms of x as well, then map and filter can be put together to express the following comprehension:

```
[e(x) | x ← Collection, p(x)] = map e (filter p Collection)
```

Map and filter act individually on elements. Map returns a new list, applying expression to each element from the list. Filter removes unwanted elements from a list, which is the task of the boolean condition shown earlier. The complex function fold is used for combining elements using a function and a list of elements, and is able of expressing both map and filter [Hut07].

More than one list at a time can be handled by the map function in some languages, provided that the lists have the same length[1]. Consequently, the last expression can theoretically

[1]Higher-order functions in Python: http://scott.andstuff.org/FunctionalPython

contain multiple collections, the same way as shown further above, given that *e* and *p* act in terms of x_1, x_2, ..., x_n over equally-sized collections:

```
map e (filter p Collection1, ... Collectionn)
```

The next paragraphs look at other functional languages and try to interpret their way of handling nested loops from an imperative perspective.

5.1.3 Loops in Selected Functional Languages from an Imperative Perspective

There is no explicit looping statement in some functional programming languages. Instead, recursive function calls are used. Stack space is consumed thereby while looping, which might lead to stack overflow in certain cases. Large or variable-sized sequences are processed usually lazily, i.e. evaluation is delayed until the expression is requested [Hal09].

Apart from implying loops in an imperative way, **Scala** can also perform loops functionally. A fundamental difference to the imperative way, is that Scala first defines the items that are going to be processed, before actually processing them. The for loops are called there *for comprehensions*. Scala successfully separates the *what* from the *how* [Pol09], which is not only a problem occurring when iterating over nested loops in an imperative-style language like Java, but generally describes a central theme in computer science [Par08]. Of course, it is not the intention to solve the fundamental problems of computer science in the context of this thesis, but it is useful every now and then to pinpoint the origins of the issues faced here. The following code snippet shows how Scala accordingly separates the process of determining which (*what*) items exactly to work with, from the process of *how* to work with the items:

```
Collection_new= for {x1 ← Collection1;
                     x2 ← Collection2 if(x1==x2)} yield x1
for (y ← Collection_new)
  e(y)
```

The code within the brackets of the for loop represents actually a generator. Additionally, it is possible to process multiple generators simultaneously in the for loop. They are not nested loops but rather *nested closures*, each generator is passed namely as a closure. Subsequently, the closure is executed for each selected item. The closure itself packages everything located to the right of the generator, which includes any later generators. The generators consequently form a kind of a chain reaction. The first left-most generator picks the matched items in the collection, and then executes the closure which calls the second generator to perform its looping, and so on, until finally the last most-right generator is reached. In the final round, the last generator executes its closure, if the the predicate is met, which runs the target code [Pol09]. The *yield* clause returns a collection containing all values of x_1, which are specified in the predecessor for comprehension that passed the boolean expression. Afterwards, each expression *e(y)* is applied to each element *y* of the newly formed collection, in an imperative-style for loop [Par08].

The **.NET languages** make use of LINQ to enable functional support. List comprehensions were already depicted through the *from-where-select* idiom in 2.1.1. Similarly, LINQ prepares first the collection to be iterated, selecting only the relevant objects before further iteration. A conventional loop can be set afterwards to iterate over the new filtered collection. Moreover, it is technically possible to iterate over two loops at the same time provided that they are of the same type using the *Concat* operator [PR07]:

```
foreach (var x in Collection1.Concat(Collection2))
    // Return x
```

It has been mentioned in 3.1.2, that a possible LINQ-to-JPQL should take into account the functional properties of LINQ. This cannot be done on the fly, since simply Java or any of its standard extensions does not support list comprehensions, or as in the case here, iterating over multiple data collections in one loop.

Loops can be constructed in **Common Lisp** in several forms, mainly by the use of keywords like *for* and *while* or better though macros like the standard *loop* or the non-standard *iterate*. In relation to the examples shown earlier, the following structure shows a possible loop construct in Common Lisp [Sei04]:

```
(loop for x1 in Collection1
    for x2 in Collection2
      if (p(x1,x2))
        collect (list x1 x2)
```

Here, two lists are iterated in parallel and a list is constructed as a result after passing the if condition. Whenever there is a need for adding new features to Common Lisp, programmers can define their own macros within minutes. This is exactly what Latendresse has done in his work [Lat07]. He presented a simple mechanism to compile list comprehensions within the loop facility and other fundamental constructs of Common Lisp. His goal was to avoid stack overflow, and primarily to write succinct code. The functional nature of Common Lisp and the possibility to use macros enables the integration of comprehensions easily, unlike the situation in most imperative languages. The opportunities of providing Java with better iteration mechanisms will be investigated in the next section.

5.2 Implementations of Effective Iteration Techniques

It has been mentioned already that Java does not support list comprehensions or higher-order functions standardly, but why actually? Comprehensions and higher-order functions pass other functions, whereas Java simply does not allow passing methods as parameters. A method in Java takes rather an interface as a parameter, whereas the interface is specified to contain a certain method. Hence, when a method that defines interfaces as parameters is called, an object of the class that implements that interface will be passed. The class can be anonymous, implementing the method first at the point of call [Jar05].

The following paragraphs examine the possibility of adding functional features to Java, mainly focusing on two approaches: The feasibility of utilizing higher-order functions and closures, and the capabilities of Java of integrating list comprehensions. The goal remains the same, i.e. to query object populations more efficiently, even without depending on the optimization techniques imposed collectively by RDBMS and ORM, but rather to be contingent upon simple effective methods that avoid moving big chunks of the database in or out of the memory.

5.2.1 Higher-order Functions and Closures

Java targets including functional qualities in the long run, e.g. JSR 223[2] and JSR 241[3]. The use of some non-standard libraries, like Lisp[4] and closures[5], enable further support. Integrating more functional properties into Java might leverage querying and iterating capabilities over large data sets. Having said that does not minimize the seriousness of the difficulties in realizing such an approach *standardly*.

[2] Jaskell as part of scripting JSR for Java: http://jcp.org/en/jsr/detail?id=223
[3] The Groovy Programming Language: http://jcp.org/en/jsr/detail?id=241
[4] Jatha - Common LISP library in Java: http://jatha.sourceforge.net/
[5] Closures for Java: http://www.javac.info/

There are number of workarounds for implementing higher-order functions in imperative languages similar to those described above. As a matter of fact, some are already part of the language. Whenever a method call is enclosed within an anonymous inner class for execution, a closure is produced indirectly, for instance:

```
Employees emp = new Employees(){public void fire() {cleanOffice();}};
```

Another eminent example of higher-order functions and closures are the visitor patterns, reviewed in section 3.1.3. Also, some methods within the class `java.utils.Collections` are considered higher-order functions[6]. Nevertheless, such functionalities are still weaker, might cause additional run-time overhead, and moreover produce more syntactic cruft compared to pure functions [Kfo08].

The functional code is obviously more straightforward. The advantage of using functional style lies in the fact that it is scalable to cope with more sophisticated computations, keeping the code in the same time succinct. To the contrary of the Java code and especially when handling data collections, it gets more verbose than queries and filtering conditions increase. Programmers are left with the task of understanding, and especially verifying, the code, which is even for experts not easy. Altenburg shows ways of implementing such functions [Alt07], and Kfoury explains the accrued complications of the transition from imperative programming to functional programming [Kfo08]. Belapurkar gives an idea of how to use closures and higher-order functions to write well-structured Java [Bel04].

Compared to higher-order functions, the creator of Python van Rossum admitted that list comprehensions are better off and have more advantages[7]. Therefore, more attention has been given to study ways of implementing list comprehensions instead.

5.2.2 List Comprehensions

There are two basic ways to implement list comprehensions: Greedily or lazily. The first way computes the whole list of the comprehension expression on the spot, returning ArrayList-similar structure with the results. The lazy way returns a similar structure to an Iterable that will do all the work later when it is asked to do so, as introduced earlier in section 5.1.3. In Python, it is the difference between comprehensions using square brackets and those using parentheses. The greedy approach is what most standard programmers expect to happen. It would have more predictable exception semantics, and theoretically should be easier to implement[8]. There are some obscure advantages in the lazy approach though. Values in such a comprehension are specified *to be used*, and stand ready to be processed. Nothing will be executed: No query or iteration is being done in the background when compiled and run. A collection of the sought records (only male employees, departments located in Munich) is at the end returned, and hence iterated without burdening the memory with redundant operations that might not be needed anyways [Par08]. Supposing that the fourth prime number greater than 1,000,000 should be calculated out of an infinite list of numbers starting at 1,000,000. Calculations cannot be performed greedily, it only works if that list is generated on demand, lazily [Hal09]. The calls are namely interpreted at the JVM level into method calls, each needing a stack frame, which will soon consume the JVM stack and hence cause stack overflow [LY99]. The lazy approach can be seen as streams, lists whose elements are being conceived as they are requested, not before [Hal09]. The typical imperative way of Java and co. showed its flaws: Iterate over *all* objects, and return those passing the if condition. The only available non-standard implementation of comprehensions will be now discussed, followed by other proposals depending solely on Java norms.

[6] Description of class collections on J2SEE: http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collections.html

[7] The fate of reduce() in Python 3000: http://www.artima.com/weblogs/viewpost.jsp?thread=98196

[8] Special thanks goes to Lawrence Kesteloot, developer of list comprehensions under the Kijaro project, for his constructive help regarding comprehensions generally and especially Kijaro's implementation.

Kijaro's List Comprehensions

There is only one available implementation of comprehensions in Java, approached by the open source project *Kijaro*[9]. It is an outside-of-the-box solution, utilizing non-standard qualities, made primarily for demonstration purposes to show the benefits of comprehensions for the Java community, and to discuss possibilities of their integration. Kijaro tend to add additional functional features to the OpenJDK javac compiler, such as first class methods, properties, list comprehensions, anonymous parameters, and others. Due to the rigorous specification and review process of adding new features to Java, the integration of Kijaro or parts of it, was not adopted in any of the proposed JSRs for Java 7[10] [11] [12]. Kijaro is seen by its opponents as an alternative language for the JVM, comparable to Scala or Groovy[13]. In order to deploy the *incomplete* code examples of Kijaro, an extended OpenJDK compiler has to be downloaded and installed locally. The compiler can be then used instead of the IDE compiler, Sun's or OpenJDK's Java compiler. The IDE compile time auto-completion assistant will be no more available when used separately. There is a possibility to integrate the Kijaro-compiler into the Eclipse IDE using Apache Ant[14]. The python-style syntax of the implementation has the following form:

```
Iterable<T> Collection_new = [ e(x) for T x : Collection if p(x)];
```

p remains the predicate, which is deployable optionally, and the $x \leftarrow$ *Collection* of the Haskell-like comprehensions is here *for T x : Collection*. The expression in the for loop has to be evaluated to a Java Iterable or an array, and the type of the identifier T has to be explicitly stated. There is no "|" to separate the expression from the qualifiers. Finally, the result expression determines the type of the whole comprehension, which is handed over to $Collection_{new}$, being either an Iterable or an array as well. The if clause can hold more than one condition (using & or |), and the source collection can be a generator. Replacing brackets by parentheses makes the resulting new collection itself a generator.

Apart from not complying with standard Java, it is not possible to loop over more than one collection in one loop. Although Iterables are considered partly more memory efficient, but they are lazily-evaluated lists[15]. Return types of the comprehensions should be adjusted to include also list and collection types. This will spare from casting results, especially if the source expression is a collection or a list, and if there is no filtering. Additionally, it is not possible to find the type of the array at runtime. The iterators should be parameterized to maintain type safety, and should coincide with the type of the returned result. Type inference of the identifier can be imposed from the type parameter of the for expression. Furthermore, it would be practical if all collection implementations have a constructor, which takes an Iterable as parameter. Truth to be told, the level of abstraction is enhanced when programming with comprehensions to an untypical Java-like level. More information on how to run Kijaro along with the other demonstrations are found on the appendix B.1.

Self-Implied Simple List Comprehensions Using Anonymous Java Classes

It has been repeatedly emphasized that most of the available implementations, supporting functional capabilities generally in Java and specifically in loops, are underscored with non-standard features. On one side, they provide practical solutions, mostly with unclouded short codes, close to the real functional functions or list comprehensions. On the other side, such

[9] Kijaro on java.net: https://kijaro.dev.java.net/
[10] A compact list of the proposed features in Java 7 with further links: http://tech.puredanger.com/java7/
[11] Small language changes for JDK 7: http://mail.openjdk.java.net/pipermail/
[12] Project Coin: http://openjdk.java.net/projects/coin/
[13] Discussion on Eclipsezone: http://www.eclipsezone.com/
[14] Apache Ant: http://ant.apache.org/
[15] Discussion on the features of Kijaro's list comprehensions: http://markmail.org/message/kpnsgxp5pfwjopyk

solutions *outside the box* are undesirable and must conform with the current or at least the next Java specifications. Therefor, a simple *out of the box* solution requiring no additional libraries and depending solely on available properties, is needed. Test code has been implemented within the scope of this thesis to demonstrate the case, but admittedly, and due to the lack of time, not further developed as would be necessary in a real application.

Since Java does not include a function type, anonymous classes are stimulated, which creates a lot of syntactic sugar as part of the deal. The implementation consists of one interface and two classes. The interface `Comprehend<T,S>` has a boolean method `join` that joins objects of type *T* and *S*:

```
public interface Comprehend<T,S>
  { boolean join(T out, S in); }
```

The class `ComprehendIns<T,S>` implements the `join` method:

```
public boolean join(Object out, Object in){
  return (out == null||in==null)
    ? false
      : myClass.isAssignableFrom(out.getClass());
}
```

Whereas the class `ComprehendUtil` likewise implements the method `comprehend` which iterates over the source collections of type *T* and *S*, and returns a filtered list back of type *T*. The application can deploy the comprehensions in two phases. The first phase prepares the lists in the following manner:

```
Comprehend<DeptManager, DeptEmp> result =
  new Comprehend<DeptManager, DeptEmp>() {
  public boolean join(DeptManager m, DeptEmp p){
    return ( //if conditions built upon m and p
    ));
  }};
```

The if conditions are those criteria to be matched, before accepting the objects. Here, `result` is of type DeptManager. Running the previous code will not burden the memory with *any* processing effort, the same way list comprehensions are implemented in most languages, i.e. lazily. Once the result is put in a loop, the above code will be computed in the second phase:

```
for (DeptManager m :
      ComprehendUtil.comprehend(result, qDeptManager, qDeptEmp)){
  // Print names
}
```

qDeptManager and qDeptEmp are those collections located in different memories. Programmers will not be delighted to see the anonymous code-style used. Indeed, the proposed list manipulation code is less elegant, or frankly said *ugly*, than similar implementations in functional languages, and not much better than that of nested loops. The syntactic sugar is caused mainly through `ComprehendIns` and specifying the return type. Apart from the verbosity, the original idea of $Collection_{new} = [e(x_1, x_2) \mid x_1 \leftarrow Collection_1, x_2 \leftarrow Collection_2, p(x_1, x_2)]$ is now formulated as $Collection_{new} = [\ e(p(T(x_1), S(x_2))), Collection_1, Collection_2]$, where *T* and *S* are types conforming with $Collection_1$ and $Collection_2$ respectively. The basic form of the code can be surely extended. More detailed comments and all code fragments are attached in appendix B.2.

5.3 Denotational Semantics for Loop Structures

List comprehensions proved their feasibility in the last sections as reasonable tools to cope with nested loops that are characterized for their extraneous details. The last paragraphs of this thesis explore the mathematical ties of comprehensions to calculus and generally connecting the findings researched so far with denotational semantics. Possibilities of providing a proper semantic for handling different types of collections, and describing the join loops formally might eventually lead to a better evaluation.

5.3.1 Monoid Comprehension Calculus and Algebra

Fegaras and Maier applied the monoid comprehension calculus and later the monoid algebra on object query language (OQL) queries in OODBMS [FM00]. Their proposed calculus performs operations over multiple collection types, aggregates, and quantifiers, abstracting common semantic features, and resulting in a *uniform way* of unnesting queries, regardless of their type of nesting, and finally applying the queries to the database. The related work is best exemplified by the OO database implementation, the lambda database [Feg04]. Additionally, a language extension for the monoid comprehension calculus was developed to avoid side effects, which occur when optimizing OO queries with updates [Feg99].

Monoids are algebraic structures used in abstract algebra[16]. Skimming over the details and borrowing some notations from [FM00], the following lines will review the monoid comprehension calculus and algebra to the depth needed for understanding the idea, and for onward considerations.

A monoid $\oplus$ has an identity $\mathcal{Z}_\oplus$ in association with the merge operation $M_\oplus$. Considering a set as a monoid, the empty set { } would be its identity and the set union $\cup$ its merge operation. The triple $\oplus = (T, \mathcal{Z}_\oplus, M_\oplus)$ is defined as a *monoid of type T*, and the quadruple $(T(\alpha), \mathcal{Z}_\oplus, \mathcal{U}_\oplus, M_\oplus)$ is denoted as a *collection monoid*, where $\mathcal{U}_\oplus$ is a function of type $\alpha \rightarrow T(\alpha)$. The quadruple $(T, \mathcal{Z}_\oplus, \mathcal{U}_\oplus, M_\oplus)$ is a *primitive monoid*, where $\mathcal{U}_\oplus$ is the identity function. Both collection and primitive monoids must fulfill specific axioms listed in [FM00], as well as the rules for *monoid homomorphisms*, which realize mappings from one monoid to another. Table 5.1 visualizes some examples of collection and primitive monoids.

Monoid $\oplus$	**type** $\mathbf{T}_\oplus$	$\mathcal{Z}_\oplus$	$\mathcal{U}_\oplus$(a)	$\mathbf{M}_\oplus$	**C/I**
Collection Monoids					
list	list(α)	[ ]	[a]	++	
set	set(α)	{ }	{a}	$\cup$	CI
bag	bag(α)	{{ }}	{{a}}	$\uplus$	C
Primitive Monoids					
sum	int	0	a	+	C
some	bool	false	a	$\vee$	CI
all	bool	true	a	$\wedge$	CI

Table 5.1: Examples of collection and primitive monoids: The last column determines whether commutativity (C) and idempotence (I) apply to the monoid. $\oplus$ is commutative if $\forall\, x, y \in$ T: merge (x, y) = merge (y, x) , and is idempotent if $\forall\, x \in$ T: merge $(x, x) = x$ [FM00].

Most importantly, $\oplus \preceq \otimes$ determines the partial order between monoids, according to their commutativity and idempotence (see table 5.1). Order is arranged between lists, bags and sets as $++ \preceq \uplus \preceq \cup$ respectively, since $\cup$ is both commutative and idempotent, $\uplus$ only commutative, and ++ is neither commutative nor idempotent [FM00]. If a mapping from lists (monoid $\otimes$) to bags (monoid $\oplus$) is considered, defining *Collection* as a list, and *e* as a

[16] Article about monoids on Wikipedia: http://en.wikipedia.org/wiki/Monoid

function taking an element x of *Collection* and returning a bag *e(x)*, then the operation can be constructed with some inspiration of table 5.1 in the following way:

$$hom[++, \uplus](e)Collection = hom^{\oplus \to \otimes}\ (e)$$

Therewith, [ ] of lists in *Collection* is replaced by {{ }}, ++ by $\uplus$, and the list [x] by *e(x)*.

The significance of monoid homomorphisms can be boiled down to the fact that it performs arbitrary divide-and-conquer technique in applying e to the elements of *Collection*, which leads to minimal ordering and division constraints, and eventually to maximal performance opportunities. On the other hand, monoid homomorphisms are signed with semantic restrictions, not all produced homomorphisms are well-formed [RT08], e.g. $\cup \not\leq ++$.

Monoid comprehensions are based on monoid homomorphism, having the form $\oplus$ { $e \mid e_1$ ... e_n}, or short $\oplus$ { $e \mid L$ }, e represents the expression to be executed if the qualifiers e_1 ... e_n apply, and L in the short form embodies a list of qualifiers, which can be empty too. If L is empty, then $\oplus$ { $e \mid$ } = $\mathcal{U}_{\oplus}(e)$, and if L is not empty, then there will be two cases. First:

$$\oplus \{\ e \mid x \leftarrow Collection, l\ \}$$
$$= \text{hom}^{\oplus \to \otimes}\ (\lambda x.\oplus \{\ e \mid l\ \})\ Collection$$

l is a comma separated list described in terms of $x \leftarrow$ *Collection* and of the predicate p. Function $\lambda x.\oplus$ { $e \mid l$ } is applied to every element x of *Collection*. The final result will be accumulated by $M_{\oplus}$. Second:

$$\oplus \{\ e \mid p, l\ \} = \text{if } p \text{ then } \oplus \{\ e \mid l\ \} \text{ else } \mathcal{Z}_{\oplus} \text{ [GS96]}$$

Which is actually the same way explained earlier when handling list comprehensions. By recalling the example [x*$x \mid x \leftarrow$ [1,2,3], even(x)], only those elements passing the predicate are given out, otherwise nothing, i.e. $\mathcal{Z}_{\oplus}$. Table 5.2 illustrates the use of the monoid comprehension in a number of examples, along with a possible interpretation in Java. Nested loops are the product of any multi-generator comprehension $\oplus${ $e \mid x_1 \leftarrow Collection_1$, ..., $x_n \leftarrow Collection_n$ } [GS96], similar to the following form:

```
result = Z⊕;
  for (x1 : Collection1)
    ...
     for (xn : Collectionn)
      result = M⊕(result, U⊕(e));
return result;
```

The situation now is almost even to what imperative languages would have done when iterating over nested loops. The result would be first initialized and then assigned the join result after passing the predicate. Generally, comprehensions having the form $\oplus${ $e \mid x_1 \leftarrow Collection_1$, ..., $x_n \leftarrow Collection_n$, $p(x_1,...,x_n)$ } would result in nested loops with depth n and having the predicate p in terms of x_1,...,x_n [GS96].

The *monoid comprehension calculus* is defined according to specific syntactic forms and typing rules stated in [FM00] as well. The main syntactic forms are re-listed in the appendix C for fast reference. Many similarities with the syntax utilized in imperative languages can be deduced, e.g. *if* e_1 *then* e_2 *else* e_3 is equivalent to e_1 *?* e_2 *:* e_3, $\mathcal{Z}_{\oplus}$ to initializing a list with a new ArrayList(), and $\mathcal{U}_{\oplus}$(e) to adding one element to the list. DeptManager and DeptEmp are considered lists that are joined now in terms of monoid comprehension calculus, with a predicate according to the department number:

$$++ [(d, e) \mid d \leftarrow DeptManager, e \leftarrow DeptEmp,$$
$$\text{DeptManager.getDeptNo()=DeptEmp.getDepartments().getDeptNo()}]$$

DeptManager.getDeptNo() is a record projection, which stands for *e.A*, e being the record and *A* the attribute, and the = sign between the two called methods is a primitive binary function.

Expression	Monoid Comprehensions	Possible pseudo-code in Java

join(p)(X, Y)	$\cup$ { $\langle a, b\rangle$ | $a \leftarrow X, b \leftarrow Y, p(a, b)$ }	`for (a:X)`
		`  for (b:Y)`
		`  if(a,b)`
		`  return a,b;`

filter(p)(X)	$\cup$ { e | $e \leftarrow X, p(e)$ }	`for (e:X)`
		`  if(e)`
		`  return e;`

$X \cap Y$	$\cup$ { e | $e \leftarrow X, e \in Y$ }	`for (e:X)`
		`  if (Y.contains(e))`
		`  return e;`

X - Y	$\cup$ { e | $e \leftarrow X, e \notin Y$ }	`for (e:X)`
		`  if (!Y.contains(e))`
		`  return e;`

$a \in X$	$\vee$ { $a = e$ | $e \leftarrow X$ }	`for (e:X)`
		`  if (e)`
		`  return true;`

length(X)	+ { 1 | $e \leftarrow X$ }	`X.size()`

Table 5.2: Examples of monoid comprehensions: The first two columns represent some expressions and their corresponding meaning in monoid comprehension [FM00]. The last column lists a possible Java interpretation, showing closeness with monoid comprehensions.

Rewriting Rules and Translation into Monoid Algebra

The monoid comprehension calculus is brought into canonical form through a normalization algorithm studied in [Feg94]. The algorithm consists of a number of rewriting rules, aiming to produce fewer intermediate data structures, and more importantly to avoid nesting of inner comprehensions. There are two important rules. First:

$$\oplus \{\ e_1 \mid x_1 \leftarrow \mathit{Collection}_1, \boldsymbol{y} \leftarrow \otimes \{\ \boldsymbol{e_2} \mid \boldsymbol{x_2} \leftarrow \boldsymbol{Collection}_2\ \}\}$$
$$= \oplus \{\ e_1 \mid x_1 \leftarrow \mathit{Collection}_1, \boldsymbol{x_2} \leftarrow \boldsymbol{Collection}_2, \boldsymbol{y} \equiv \boldsymbol{e_2}\ \}$$

The inner monoid comprehensions $\otimes$ is transformed into multi-generator comprehension, defining the variable y as a synonym for the expression e_2 in all following qualifiers. Second:

$$\oplus \{\ e_1 \mid x_1 \leftarrow \mathit{Collection}_1, \vee \{\ \boldsymbol{p} \mid \boldsymbol{x_2} \leftarrow \boldsymbol{Collection}_2\ \}\}$$
$$= \oplus \{\ e_1 \mid x_1 \leftarrow \mathit{Collection}_1, \boldsymbol{x_2} \leftarrow \boldsymbol{Collection}_2, \boldsymbol{p}\ \}$$

The algorithm rearranges all operators in one level, enabling free movement of predicates between inner and outer comprehensions and queries. It helps especially in cases, where a collection extracts its element from another one, like the sub-query in the SQL query `SELECT...FROM(sub-query)...WHERE...`. However, it is not allowed to iterate over an array or an instance of `java.lang.iterable`, or said in other words, the elements of a collection cannot be generally fed into another loop in the same loop structure in most imperative languages:

```
for (Departments d: (for( DeptEmp e: qDeptEmp))
```

Even if dealing only with queries, the normalization algorithm cannot unnest all forms [FM00]. The *monoid algebra* extends the work of monoid comprehensions, considering means for a better unnesting technique. On top of this, the algebra enables direct translation of queries into physical mechanism. The algebra consists of seven more or less operators, which are listed in the appendix C. Considering the example:

```
for (Employees e: qEmployees)
  for (DeptEmp d: qDeptEmp)
    if (d.getEmployees().getEmpNo() == e.getEmpNo())
      return e
```

It can be translated into monoid algebra as in:

$$\text{qEmployees} \bowtie_{d.getEmployees().getEmpNo()==e.getEmpNo()} \text{qDeptEmp} =$$
$$\Gamma_d{}^{\cup/e}\ \Delta^{\cup/<D=d,E=m>}$$

The original work makes use first of monoid calculus to normalize the queries, and then of the algebra to unnest them before evaluation can take place on the database. Moreover, [GS96] discussed the relationship between loops and queries in terms of monoid calculus and algebra. It has been shown that monoid comprehensions correspond to similar structures in Java, but the situation discussed here is from a wider view rather different. The calculus is deployed primarily to simplify loops of elements iterating over the same collection, or deduced from one major collection. Here, there are two object populations having different sources. On the other hand, monoid algebra can be explicitly coded as operators into Java, refraining from the restrictions of not having comprehensions in the first place. Upon asking the authors of [FM00] about the feasibility of applying the monoid comprehension calculus on the designed scenario, they expressed their doubts and advised using other methods[17].

[17] Fegaras wrote on July 29, 2009: "*I think you can do this better using other methods.*"

5.3.2 The Need of Denotational Semantics

Broadly speaking, many programming languages and specially database languages lack denotational semantics [Mit03]. The last section gave an example of the feasibility of having solid axiomatic and operational semantics, like the monoid comprehension calculus in the case of list comprehension and moreover for OQL. The last parts of this thesis discuss the need of such formalizations to leverage the work of programming languages and the designed application to be equipped with better tools for diagnosing the problems and crafting solutions accordingly.

Alves-Foss and Lam presented dynamic denotational semantics of Java that cover many aspects of the base language [AFL99]. Related to the OO model discussed here, the *category theory* represents another approach for formalization. Richta and Toth argue that it is possible to deploy the category theory to add denotational semantics not only to OO models, but also to relational and hierarchical ones [RT08]. Category theory is known in mathematics, abstracting sets and functions to objects linked in diagrams by morphisms. The design of several functional programming languages is said to be inspired by the category theory [BW90].

Both category theory and OO programming languages, like Java, have many similarities. Categories are nothing but collections of objects, and morphisms correspond to the relationships between these collections. More specifically, the entities handled here can be seen as sets that are abstracted to objects, interacting with each other according to functions. Referring to the discussed scenario, female and male employees are considered sets that are joined by a union to form a new set: $Employees = Employees_{female} \cup Employees_{male}$. Functions can be defined to transform from these sets to other ones $Employees_{filtered} = \{\ x \mid e(x) \mid x \in Employees\ \}$, similar to the idea of list comprehensions. To assert their membership in a certain category, objects of a collection must fulfill the axioms listed in [RT08]. An OO programming language would traditionally assign a class structure to each object of a different category, and define its extension and inheritance properties. Categories are represented as a generalization of those classes, namely as interfaces.

```
class Employees_female extends Employees { ... };
```

By default, the classes must implement all the methods stated in the interface. If the methods defined in the interface represent the membership axioms, implementing them in the class means indirectly being a member of the category. Languages have different interpretations of how to apply category theory in practice. In Haskell, being one of the closest, functions are first defined to be applied to certain category class, and once the category supports these functions, then they are considered members of this category[18]. Stay explores the possibilities of integrating the category theory into Java [Sta07]. More interesting is the work of Liebmann, he deploys the category theory to solve parallel numerical algorithms using C++ [Lie06].

Denotational semantics are used in most cases for reasoning about programs, languages optimizations, and static analysis methods. Setting new semantics for a language should take into account future extensions to include new features [Mit03]. There are good opportunities for connecting database models with the category theory as the authors of [RT08] did, to associate Java with denotational semantics as started by [AFL99], which might lead to solve complex problems as it has been already achieved using an OO language [Lie06].

5.4 Summary

The goal of this chapter was not to discuss functional languages versus Java, but rather to point at the problems emerging when naive loops are implemented generally in imperative

[18] Category theory with Haskell on Wikibooks: http://en.wikibooks.org/wiki/Haskell/Category_theory

languages, and more generally to explore the possibilities of having functional properties for query processing. The ideas mentioned and the proposed solutions were based on the fact that Java and its JPA were used from the beginning to realize querying over persistent OO data models. The classical imperative way of treating loops would be to take a collection, iterate over all objects, and return certain values after passing a specific condition. The functional way instead returns values out of a collection that match the condition, and then iterates over the filtered collection, separating the what from the how. Iterating over lists to perform main-memory joins operations can be done much simpler using list comprehensions, and hence adds functional qualities when querying over databases and object collections originating from different data sources. It will be possible then to manipulate object populations directly on the application level, without having to transport the data in or out of the memory for any changes. The Java implementation of list comprehensions achieved so far was demonstrated through the Kijaro project, which brings along non standardized concepts, and through a self-implied solution using anonymous classes, which adds syntactic sugar and lack full implementation.

A relationship between monoid comprehensions and nested loops has been triggered. Further on, the monoid comprehension calculus was explained. The calculus provides list comprehensions with solid mathematical background. The original work handled queries, and discussed possibilities of unnesting them through the use of comprehension and algebra. Here, the queries are already executed, and there are two or even more object populations, which need to be iterated to perform some operations. The monoid comprehension calculus cannot be used to simplify the nested loop produced in this scenario. The normalization algorithms were designed primarily to unnest queries from other queries operating on the same data source. The authors of the monoid comprehensions confirmed that their calculus is not feasible to handle the nested loops here. There is a possibility, however, to implement the algebra operators to be able to handle nested loops more efficiently and automatically. Finally, the benefits of adding denotational semantics to programming languages and especially to OO data models were discussed, referring to success stories in other OO languages and linking to possible potentials for future considerations.

Chapter 6

Conclusion and Future Work

This thesis encompassed a number of concerns that are characterized for being relatively noticeable in the database research. A slightly different approach has been taken in addressing the topics, more or less pragmatically, trying to emulate everyday situations in a simplified manner. The problems were first impartially reviewed and then traced back to root circumstances under which they were developed, and possible solutions were examined systematically. The following lines summarize the outcome of this work, followed by a list of suggestions for future investigations.

6.1 Conclusion

As mentioned in the introduction, this work is split into two main parts, one dealing with the possibilities of querying persistent data type-safely and functionally within the JPA framework and one dealing with the feasibility of processing different object populations after query evaluation. Several interesting new results have been found.

Placing high bets on JPQL is risky. Neither the old version, nor the new one can be in line with LINQ. In section 2.2, the new published specification of JPA 2.0 clarified two important points in relation to the research process. First, JPQL will not change a lot. Second, the Criteria API of JPA 2.0 will do everything JPQL does. Additionally and with the meta-model API, the Criteria API will preserve type safety and perform dynamic query composition [Mic09]. It is a matter of time until a full-featured compiler and vendor support will be available. A look at the website of EclipseLink indicates that more than half of the assigned implementation tasks are already done[1]. This is the main reason for putting the idea of translating LINQ into JPQL aside. Type safety queries can be achieved with the new version of JPA. Nevertheless, performing functional queries is dependent on the underlying programming language. The missing functional traits of Java can be compensated partially, as pinpointed in 5.2.1 and through the integration of list comprehensions as started in 5.2.2.

Technical procedures for a possible translator were described in section 3.1. The thesis had the advantage of deploying interim solutions, thus, exerting more effort in shedding light upon essential problems of more importance. The emerging issues deserve further investigations, and are not related to a specific tool or language, but rather describe general phenomena. The aim of the application scenario utilized within the scope of this thesis, is to put research into practice, and to serve as a basic configuration for a possible extension to include JPA 2.0 or other data sources. The used database reflected a typical data store, and the code examples were written down to show how simple the problems can look like, to the contrary of their possible solutions.

[1] EclipseLink JPA 2.0 implementation status: http://wiki.eclipse.org/EclipseLink/Development/JPA_2.0

The second part of the work investigated data processing after query evaluation, and tried to answer the question imposed at the beginning of this work, namely how to process the produced data more efficiently. Indirectly, adding functional properties to the executed queries were pursued. Query results were returned and inserted in lists, as it would happen in most real applications. It has been indicated that optimizations can be done on objects originating from the secondary-memory. In practical scenarios, there will other objects in main-memory, which will be brought together with the secondary-based ones for further processing. Especially for those main-memory objects, there will be problems computing join operations. The formed object populations are comparable with the idea of having different data models, not only relational but also hierarchical. Iterating over all main-memory objects and then over all secondary-memory objects would lead definitely to a nested loop, consuming the resources of the system impractically. Other enhanced join methods would not solve the problem thoroughly. The complications hindering data transfer were explained by reference to the different memory architechtures. The need of obtaining a flexible way of manipulating object populations more effectively was emphasized, without having to move chunks of data in or out of the memory, and without writing long code lines. The difficulties of transporting large secondary-memory collections into main-memory were experienced and indicated coincidentally a recent bug in the ORM framework 4.1.1.

As mentioned at the very beginning of this thesis, Java is the enterprise language number one, and probably after this research, is the most language in need of rehabilitation. A key requirement in the business world is lists, and extracting data out of them. The same as in the application scenario handled throughout this thesis. The idea was simple: To iterate over large object collections more effectively than nested loops would do. Functional qualities were simply probed to iterate effectively over object results. Such experiments have stumbled across a number of inconsequential, but nonetheless amazing obstacles, at least from the view of functional programming languages. The for-each loop has indeed brought some advantages into Java, which can be seen as an additional macro to express the typical for and while loops in a neat way. Common Lisp programmers can easily add their own macros within minutes to give themselves a source-level of abstraction of encountered common patterns, as Latendresse did in [Lat07]. It would be nice to have the same ritual in Java, for example when considering comprehensions, but quite contrary to the rigorous procedure adopted by Sun. Any new language features have to go through an industry-wide expert group, which lasts normally up to one year. Finally, vendor-compilers have to get along and implement the new features. Adding functional properties into Java, like closures 3.2.2 and the python-style comprehensions 5.2.2, have failed to get acknowledged by any JSR. The language could have reached a higher level of abstraction faster in the absence of these restrictions, and probably this is the reason behind the popularity of Scala and Groovy in the last years. On the other hand, the same procedures have maintained the language from being forked and fragmented for every possible reason. Despite all that, Java is now more attractive than ever, thanks to its problems.

The next step taken was to look around, and examine how other languages tried to solve the same experienced problems, among them LINQ. List comprehensions are capable of mastering loops over different data sets. They have succinct effective *good* code, which cannot be deployed per se on Java. Their idea was narrowed down at first to that of imperative languages. The *bad* code of nested loops was replaced later by a self-implemented *ugly* code. The proposed solution is devised as a start for the use of list comprehensions in Java, solely depending on norm components. Unlike the concise more direct suggestion of Kijaro, which exerts non-standard instruments. The good news about both tools, is that they tend to separate the *what* from the *how* when iterating over collections, having the same mentality of most functional constructs.

Truth to be told, the last paragraphs of chapter five were the most interesting ones. Here, a priority was set to search for possibilities of extending the *good*, the *bad* and the *ugly* codes

by an implementable *good* code on Java, for example by unnesting the bad one and finally by deploying the ugly technique. A relationship between comprehensions and nested loops has been triggered by the authors of [FM00], which was further studied. The monoid comprehensions were linked to similar structures in Java. List comprehensions are retroactively amended by the high-level and uniform monoid comprehension calculus, which can be easily normalized. The monoid algebra is low-level, offering better query unnesting techniques, which can be directly transformed into physical algorithms. The monoid calculus and algebra were applied on one of the use cases, and determined that calculus cannot solve the problem thoroughly. Instead, the algebra could be able to unnest complex data collections into simpler ones. Finally, the role of category theory in a possible interpretation of the formed scenario. There are significant potentials and opportunities to utilize the category theory to handle different object collections uniformly.

The research groups behind the works of [FM00], [Teu06], [WPN06] and most recently [RT08] have a common goal, and that is to provide denotational semantics to some kind of a query language or more general to a database model. This is the main difference between academic and industrial approaches. The product of industry, Java, was not able to solve a simple problem in an effective way, having no solid background. The formal modeling of the research groups can handle any kind of data and should be able to analyze any future problems according to a set of rules. Admittedly, the problems defined in the first pages of this thesis seemed very easy, but soon turned not to be. However, the different issues handled were filtered and narrowed down to specific requirements. Due to the lack of time, as in any thesis work, these issues could not be brought to an end. Nevertheless, they can be researched perfectly in future works.

6.2 Future Work

The research presented in this thesis seems to have raised more questions that it has answered. These questions could not be answered in the present analysis but nevertheless should be investigated in future works:

- Future investigations should have the enhanced **Criteria API** of JPA in sight, not JPQL. The Java community will be equipped with a better mechanism to outstrip competitors in this arena, like LINQ. The knowledge gained and progress done by [Pri08] and [Wen09], can be however proceeded to perform another kind of translation, namely from LINQ to the Criteria API. All in all, the idea of translation from LINQ to a Java-based query language will not delegate the functional properties of LINQ, and will have little impact. A considerable effort of understanding software-engineering technology-specific tools such as EMF will be needed in this case. This is due to the fact that the previous work is EMF-based. Prototyping would be reasonable if it should serve as a demonstration case for further demands as [GP08] tried to do. Translating from comprehensions directly into Criteria API would be more adequate. Nevertheless, each nested list comprehension will be probably translated into one Criteria query, leading to several Criteria queries per one comprehension.

- Ways of extending the scenario handled to include **hierarchical data models** can be advantageous, and could be achieved as listed in 4.1.2. Hierarchical technologies and data sources are on the marsh, backed by the expanding web services. There are needs for querying results coming from relational and hierarchical origins. Apart from theoretical studies, practical work would preserve traces achieved so far, and hopefully leave its marks for prospective analysis. Best practices of [Teu06] would be helpful in addressing this issue. Further light should be shed upon **handling methods** of the same object structure, which are defined to operate on an object population, but not

recognized by the other one as briefly depicted in section 4.2.2. ORM solutions do not handle this matter sufficiently.

- The database community is aware of the power of functional languages, whereas the **Java** community is blindfolded with Java's advantages. The language will stay roughly the same for the next couple of years. It is not equipped with efficient language support for lists or maps, and here where others got ahead. Enhanced list processing techniques can be achieved, and would surely make a fair impression on the Java community. The thesis took the plunge and pinpointed some of the language's shortcomings from a specific angle, namely in the context of OO data modeling. That is indeed not all. A good start to support **functional properties** has been set. This work tried to use closures and anonymous classes to realize list comprehensions using solely the norms allowed. Java can still be pushed as far as possible to implement referential transparency and concurrency. The work of [Bel04] can be expanded, and iterating techniques can be theoretically enhanced through concurrent programming [ZHR+06].

- First-class constructs are finding their way from research into main-stream OO languages, as exemplified by LINQ. Java programmers are left to implement relationships in an ad-hoc fashion which results in unnecessarily complex code. Open source Java **query and list implementations**, like JaQue 3.2.2 and Kijaro 5.2.2, can be extended without having to start from scratch on the one hand. On the other hand, they lack denotational semantics. Although, integrating the high-level list comprehensions into Java might cause a level-of-abstraction mismatch, they would help performing functional queries and will lead indeed for a better iteration technique. The self-implied ugly code can be further improved in a full-scale implementation to turn hopefully to a good code.

- The data of the future will have different backgrounds (relational, hierarchical, etc.), and will be brought in for further collective processing. However, there is a need to handle them uniformly. Studies like [RT08] have an ambitious goal, adding **formal categorical approach** to different database modeling in terms of the category theory. In this way, data will be understood better and will be handled upon specific mathematical axioms. The challenge is to package the advanced research ideas into an OO database or programming language and more interestingly, into a legible easy-understanding manner. Similar to the findings of [Lie06], there are opportunities of combining the category theory with an OO language to be able to process complex computations. A possible start in Java would be the work described in [Sta07].

These suggestions can serve as a foundation for future research; research that will indeed transcend the contributions done here and negate its limitations. Further studies are needed to gain insight into *type-safe functional queries over object-oriented data models*.

Appendix A

Project Configuration Preferences

The following descriptions refer to settings applied on a machine having Linux Ubuntu 9.04 as an operating system, using Eclipse as an IDE, Hibernate as an ORM framework, MySQL as a RDBMS, and without an application server.

A.1 Preparing the JPA Project

1. Download and install Eclipse IDE, by typing on the console *sudo apt-get install eclipse* or manually from[1]. The export path should set correctly, e.g. on the console: *export JAVA_HOME=/usr/lib/jvm/java-6-sun-1.6.0.13/jre* and *export CLASSPATH=/usr/lib/jvm/java-6-sun-1.6.0.13/jre/lib*.

2. Download and install a JPA ORM vendor into Eclipse, e.g. Hibernate[2].

3. Download a RDBMS, e.g. MySQL, type on the console: *sudo apt-get install mysql-server mysql-client php5-mysql mysql-admin libmysql-java*. php5-mysql and mysql-admin are used mainly for viewing the RDB on a GUI, whereas libmysql-java is needed for the JDBC driver. Subsequently, set the username and password for accessing the RDB, e.g. *mysqladmin -u root password root*

4. Download and install RDBMS JDBC driver into Eclipse, e.g. the JDBC driver for MySQL[3]

5. Download a sample database. Employees is a large database containing up to four million records[4]. Import the database into the RDBMS, for MySQL use *mysql -uroot -proot -h 127.0.0.1 DBNAME < DUMPFILE*. If a GUI interface is used for the import, like PHPMyAdmin, then memory will be exhausted dramatically depending on the size of the imported database.

6. In Eclipse, set a database connection to the RDBMS using the JDBC driver and save it e.g. under *TFJPAconnect* and start the connection to the RDBMS.

7. Download the Dali plugin in Eclipse[5]. It enables reverse engineering and imports entities automatically from an existing RDB schema. Afterwards the persistence provider takes care of persistence and data management from the object level.

[1] Eclipse: http://www.eclipse.org/downloads/
[2] Hibernate: https://www.hibernate.org/
[3] Using MySQL With Java: http://dev.mysql.com/usingmysql/java/
[4] Sample database with test suite: https://launchpad.net/test-db/
[5] Dali: http://www.eclipse.org/webtools/dali/main.php

A.2 Setting the JPA Project

1. Create the JPA project in Eclipse, and Java packages as needed. Set the the platform property to *Hibernate*, and the connection to *TFJPAconnect*. Make sure that *the default schema is overridden to employees* in the JPA properties is checked and *annotated classes must be listen in persistence.xml* as well. Using *FPGA* as a name for the persistence unit, MySQL as the RDBMS, employees as the database name which is hosted locally, the persistence.xml file can be written as the following:

Listing A.1: The persistence.xml file

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/
       persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
       instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
       persistence http://java.sun.com/xml/ns/persistence/
       persistence_1_0.xsd">
3  <persistence-unit name="FJPA" transaction-type="
       RESOURCE_LOCAL">
4  <provider>org.hibernate.ejb.HibernatePersistence</provider>
5  <class>org.entity.employees.VFullEmployeesId</class>
6  <class>org.entity.employees.VFullEmployees</class>
7  <class>org.entity.employees.VFullDepartmentsId</class>
8  <class>org.entity.employees.VFullDepartments</class>
9  <class>org.entity.employees.TitlesId</class>
10 <class>org.entity.employees.Titles</class>
11 <class>org.entity.employees.SalariesId</class>
12 <class>org.entity.employees.Salaries</class>
13 <class>org.entity.employees.Employees</class>
14 <class>org.entity.employees.DeptManagerId</class>
15 <class>org.entity.employees.DeptManager</class>
16 <class>org.entity.employees.DeptEmpId</class>
17 <class>org.entity.employees.DeptEmp</class>
18 <class>org.entity.employees.Departments</class>
19 <properties>
20 <property name="hibernate.connection.driver_class" value="com
       .mysql.jdbc.Driver"/>
21 <property name="hibernate.connection.username" value="root"/>
22 <property name="hibernate.connection.password" value="root"/>
23 <property name="hibernate.connection.url" value="jdbc:mysql:
       //127.0.0.1:3306"/>
24 <property name="hibernate.dialect" value="org.hibernate.
       dialect.MySQLDialect"/>
25 </properties>
26 </persistence-unit>
27 </persistence>
```

   The database login data can be adjusted to match the local settings. If another RDBMS is deployed, then these definitions should be accordingly updated. After synchronizing the persistence.xml, the mapped entities will be inserted automatically after the provider element.

2. Using the Dali/JPA tools plugin, generate entities from the connection to RDBMS. Now, it is possible to use the source codes mentioned in the next chapter.

Appendix B

Source Code for Testing Purposes

B.1 Using Quaere, JaQue, and Kijaro

1. Quaere is available only as a subversion repository. Use the following commands: *svn checkout http://svn.codehaus.org/quaere/trunk/*, then let Maven install it locally by *mvn install* and making an Eclipse project out of it *mvn eclipse:eclipse*. Now, the project can be imported into Eclipse and used there.

Listing B.1: Queries with Quaere

```
1  package org.test;
2
3  import static org.quaere.DSL.*;
4  import javax.persistence.EntityManagerFactory;
5  import javax.persistence.Persistence;
6  import org.entity.employees.Departments;
7  import org.quaere.jpa.QueryableEntityManager;
8
9  public class Test_Quaere {
10  public static void main(String[] args){
11   EntityManagerFactory entityManagerFactory =
12            Persistence.createEntityManagerFactory("FJPA");
13          start1 = System.currentTimeMillis();
14
15   QueryableEntityManager entityManager =
16            new QueryableEntityManager(entityManagerFactory.
                createEntityManager());
17   // Select the department number where the department name
         is "Sales"
18   Iterable<Departments> qDepartments =
19    from("d").in(entityManager.entity(Departments.class)).
20    where(eq("d.getDeptName()","Sales")).
21    select("d");
22   // Print the result
23   for (Departments dep : qDepartments) {
24     System.out.println(dep.getDeptNo());
25   }
26  }
27 }
```

2. JaQue has a JAR, which can be downloaded directly from[1]. The following code performs a simple query over the entity Employees.

Listing B.2: Queries with JaQue

```
1  package org.test;
2
3  import static jaque.DynamicQuery.select;
4  import static jaque.DynamicQuery.where;
5  import jaque.Queryable;
6  import jaque.functions.Function;
7  import jaque.functions.Predicate;
8  import jaque.jpa.JaqueEntityManager;
9  import javax.persistence.EntityManager;
10 import javax.persistence.EntityManagerFactory;
11 import javax.persistence.Persistence;
12 import org.entity.employees.Employees;
13
14 public class Test_Jaque {
15  public static void main(String[] args){
16   EntityManagerFactory emf = Persistence.
        createEntityManagerFactory("FJPA");
17   EntityManager em = emf.createEntityManager();
18   JaqueEntityManager jem = new JaqueEntityManager(em);
19   //Select all male employees where EmpNo < 10100
20   Queryable<Employees> qEmployees = select(new Function<
        Employees, Employees>() {
21    public Employees invoke(Employees t) throws Throwable {
22     return t;
23    }
24   }, where(new Predicate<Employees>() {
25    public Boolean invoke(Employees e) throws Throwable {
26     return e.getEmpNo()<10100;
27    }}, where(new Predicate<Employees>() {
28     public Boolean invoke(Employees e) throws Throwable {
29      return e.getGender().equals("M");
30     }
31    },jem.from(Employees.class))));
32   // Print the result
33   for (Employees e : qEmployees) {
34    System.out.println(e.getEmpNo()+
35     ":␣" + e.getLastName());
36   }
37  }
38 }
```

3. Kijaro should be built locally to produce the altered version of javac. First, register on their site to get access to the repositories, then type *svn checkout https://kijaro.dev.java.net/svn/kijaro/branches/listcomprehensions kijaro –username USERNAME –password PASSWORD*. Afterwards, change directory to go the place where kijaro has been downloaded and go to /langtools/make. Now, use ANT

[1] JaQue on Google Code: http://code.google.com/p/jaque/

to build it: *ant -Dboot.java.home=$ JAVA_HOME*. The altered javac can be used now as if using the normal one.

Listing B.3: Queries with list comprehensions under the Kijaro project

```
package org.test;

public class Test_ListCom {
 public static void main(String[] args){
  // Prepare the lists
  Iterable<Employees> dm = [m.getEmployees()
    for DeptManager m: qDeptManager_MM
    if (m.getToDate().equals(todate)&
    ( m.getDepartments().getDeptNo().equals("d006")))];
  Iterable<Employees> de = [e.getEmployees()
    for DeptEmp e: qDeptEmp_SM
    if (e.getToDate().equals(todate)&
    (e.getDepartments().getDeptNo().equals("d006")))];
  // Iterate over the new lists
  for (Employees dd: dm){
   for(Employees ee: de){
    // Print results
   }
  }
 }
}
```

B.2 Self-Implied Simple List Comprehensions Using Anonymous Java Classes

1. Interface `Comprehend<T,S>` having only one method that joins (accepts) other objects of the same types:

Listing B.4: Interface Comprehend for processing list comprehensions

```
package org.lc;

public interface Comprehend<T,S>
{
    boolean join(T out, S in);
}
```

2. `ComprehendIns` implements the interface Comprehend and its join method. The `join` method takes two objects as an argument and returns the boolean value of `isAssignableFrom(obj.getClass())`, which tests whether the type represented by the specified Class parameter can be converted to the type represented by this Class object via an identity conversion or via a widening reference conversion. More can be found on [GJSB05].

Listing B.5: Class ComprehendIns for implementing the interface Comprehend

```
package org.lc;

```

```
3  public class ComprehendIns<T,S> implements Comprehend<T,S>
4  {
5   private final Class<? extends T> myClass;
6   public ComprehendIns(Class<? extends T> myClass)
7   {
8    this.myClass = myClass;
9   }
10  public boolean join(Object out, Object in)
11  {
12   return (out == null||in==null)
13   ? false
14    : myClass.isAssignableFrom(out.getClass());
15  }
16 }
```

3. The class `ComprehendUtil` has one method that takes two input object collections as arguments, one of the interface<*T*,*S*> Comprehend, and two of iterable source collections of types *T* and *S* respectively. A result list of type *T* is produced. The loop filters the input sources and passes only those objects that will be accepted by the filter:

Listing B.6: Class ComprehendUtil for list processing

```
1  package org.lc;
2
3  import java.util.LinkedList;
4  import java.util.List;
5
6  public class ComprehendUtil {
7   public static <T,S> List<T> comprehend
8   (
9    Comprehend<T,S> filter,
10   Iterable<T> source1,
11   Iterable<S> source2
12  )
13  {
14   List<T> result = new LinkedList<T>();
15   for (T t : source1)
16   {
17    for (S s : source2)
18    {
19     if (filter.join(t,s)){
20      result.add(t);
21     }
22    }
23   }
24   return result;
25  }
26 }
```

4. Two object populations are first emulated and then joined together to be filtered upon certain conditions. Therewith, the lists will be prepared, and finally looped as in the following program:

Listing B.7: The main class to demonstrate processing using list comprehensions

```
1  package org.lc;
2
3  import static jaque.DynamicQuery.select;
4  import jaque.Queryable;
5  import jaque.functions.Function;
6  import jaque.jpa.JaqueEntityManager;
7  import java.util.ArrayList;
8  import java.util.List;
9  import javax.persistence.EntityManager;
10 import javax.persistence.EntityManagerFactory;
11 import javax.persistence.Persistence;
12 import org.entity.employees.DeptEmp;
13 import org.entity.employees.DeptManager;
14
15 public class Main {
16  public static void main(String[] args) {
17   // JPA entity manager
18   EntityManagerFactory emf = Persistence.
        createEntityManagerFactory("FJPA");
19   EntityManager em = emf.createEntityManager();
20   // Jaque entity manager
21   JaqueEntityManager jem = new JaqueEntityManager(em);
22   // Timers and counter
23   final java.sql.Date todate = java.sql.Date.valueOf("
        9999-01-01");
24   try {
25    // Prepare object populations
26    // 1. Main-memory collection
27    List<DeptManager> qDeptManager_MM = new ArrayList<
         DeptManager>();
28    List<DeptManager> DeptManager_MM_temp = em.createQuery
29     ("SELECT dept_manager FROM DeptManager dept_manager").
          getResultList();
30    if(DeptManager_MM_temp.size()!=0){
31     for (DeptManager emp : DeptManager_MM_temp){
32      qDeptManager_MM.add(emp);
33     }
34    }
35    // 2. Secondary-memory collection
36    Queryable<DeptEmp> qDeptEmp_SM = select(new Function<
         DeptEmp, DeptEmp>() {
37     public DeptEmp invoke(DeptEmp t) throws Throwable {
38      return t;
39     }
40    }, jem.from(DeptEmp.class));
41    // List comprehensions
42    // 1 List preparations
43    Comprehend<DeptEmp, DeptManager> result =
44     new Comprehend<DeptEmp, DeptManager>() {
45      public boolean join(DeptEmp p, DeptManager m){
46       return(
```

```
47          m.getDepartments().getDeptNo().equals("d006")&&
48          p.getDepartments().getDeptNo().equals("d006")&&
49          m.getToDate().equals(todate)&&
50          p.getToDate().equals(todate)
51         );
52        }
53     };
54     // 2 List iteration
55     for (DeptEmp pp:ComprehendUtil.comprehend(result,
           qDeptEmp_SM, qDeptManager_MM)){
56      System.out.println(
57       pp.getEmployees().getEmpNo()+": "+
58       pp.getEmployees().getFirstName()+ " "+
59       pp.getEmployees().getLastName()+", "+
60       pp.getDepartments().getDeptNo());
61     }
62     em.close();
63    }
64    catch (Exception e) {
65     System.out.println("Exception occured reading the file.");
66    }
67   }
68 }
```

Appendix C

Monoid Comprehension Calculus and Algebra

Form	Meaning
NULL	null value
c	constant (string, int, bool, ...)
v	variable
e.A	record projection (attribute A of record e)
if e_1 then e_2 else e_3	if-then-else statement
e_1 op e_2	a primitive binary function (+,=,<,>)
$\mathcal{Z}_\oplus$	zero element
$\mathcal{U}_\oplus$(e)	singleton construction
$e_1 \oplus e_2$	merging
$\oplus\{e \mid q_1, , q_n\}$	comprehension

Table C.1: The main forms of monoid comprehension calculus: *e, e_1, ..., e_n* are terms of the calculus and q_1, , q_n are qualifiers [FM00].

Operator	Algebra	Calculus
join	X $\bowtie_p$ Y	{ (v,w) | v ← X, w ← Y, p(v,w) }
selection	σ_p(X)	{ v | v ← X, p(v) }
unnest	μ_p path (X)	{ (v,w) | v ← X, w ← path(v), p(v,w) }
reduce	$\Delta_p{}^{\oplus/e}$ (X)	$\oplus$ e(v) | v ← X, p(v)
outer-join	X $\bowtie_p$ Y	{ (v', w) | v ← X, w ← if ∧{ ¬p(v,w') | v ≠ NULL, w' ← Y } then [NULL] else { w' | w' ← Y, p(v,w') } }
outer-unnest	μ_p path(X)	{ (v',w) | v ← X, w ← if ∧ { ¬p(v,w') | v ≠ NULL, w' ← path(v) } then [NULL] else { w' | w' ← path(v), p(v,w') } }
nest	$\Gamma_{p/g}{}^{\oplus/e/f}$ (X)	{ (v,⊕{ e(w) | w ← X, g(w) ≠ NULL, v ≐ f(w), p(w) }) | v ← Π_f (X) }

Table C.2: Monoid algebra [FM00].

Bibliography

[AFL99] Jim Alves-Foss and Fong Shing Lam. Dynamic denotational semantics of java. In *Formal Syntax and Semantics of Java*, pages 201–240. Springer Verlag, Berlin, Germany, 1999.

[Alt07] David Altenburg. Enumerate, Map, Filter, Accumulate. `http://gensym.org/2007/4/7/enumerate-map-filter-accumulate`, April 2007.

[AP07] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Foundation Press, Inc., Mineola, NY, USA, 2007.

[Bel04] Abhijit Belapurkar. Functional Programming in the Java Language. `http://www.ibm.com/developerworks/java/library/j-fp.html`, July 2004.

[Bil02] Robert W. Bill. *Jython for Java Programmers*. Sams Publishing, London, United Kingdom, 2002.

[BK06] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[CB04] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation and Management (International Computer Science Series)*. Addison Wesley, London, United Kingdom, 2004.

[Feg94] Leonidas Fegaras. A Uniform Calculus for Collection Types. Technical report, Oregon Graduate Institute of Science & Technology, Beaverton, OR, USA, 1994.

[Feg99] Leonidas Fegaras. Optimizing Queries with Object Updates. volume 12, pages 219–242. Kluwer Academic Publishers, Hingham, MA, USA, 1999.

[Feg04] Leonidas Fegaras. λ-DB. `http://lambda.uta.edu/ldb/doc`, December 2004.

[FM00] Leonidas Fegaras and David Maier. Optimizing Object Queries using an Effective Calculus. volume 25, pages 457–516. ACM, New York, NY, USA, 2000.

[Gar08] Miguel Garcia. *On the Formalization of Model-Driven Software Engineering*. PhD thesis, Institute for Software Systems, Hamburg University of Technology, Hamburg, Germany, 2008.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, The Netherlands, 2005.

[GMRS09] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. FERRY: Database-Supported Program Execution. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 1063–1066, Providence, RI, USA, 2009.

[GP08] Miguel Garcia and Rakesh Prithiviraj. Rethinking the Architecture of O/R Mapping for EMF in terms of LINQ. In *Eclipse Modeling Symposium at Eclipse Summit Europe*, Stuttgart, Germany, 2008.

[GS96] Torsten Grust and Marc H. Scholl. Translating OQL into Monoid Comprehensions - Stuck with Nested Loops? Technical report, University of Konstanz, Department of Mathematics and Computer Science, Konstanz, Germany, 1996.

[Hal09] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, Raleigh, NC, USA, 2009.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, United Kingdom, 2007.

[Jar05] Duane J. Jarc. *Java Programming & Projects (JDK 5th Edition)*. Dreamtech Press, New Delhi, India, 2005.

[Kfo08] Assaf Kfoury. Lecture Notes on Concepts of Programming Languages. `http://www.cs.bu.edu/faculty/kfoury/CVS-Working-Files/CS320-Fall08/`, September 2008.

[Kle08] Scott Klein. *Professional LINQ*. Wrox Press Ltd., Birmingham, United Kingdom, 2008.

[KS06] Mike Keith and Merrick Schincariol. *Pro EJB 3: Java Persistence API*. Apress, Berkely, CA, USA, 2006.

[Lat07] Mario Latendresse. Simple and Efficient Compilation of List Comprehension in Common Lisp. In *In the Proceedings of the International Lisp Conference*, pages 125–130. Cambridge, United Kingdom, 2007.

[Lie06] Manfred Liebmann. Category Theory and the Design of Parallel Numerical Algorithms. Max Planck Institute for Mathematics in the Sciences, Leipzig, Germany, 2006.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Prentice Hall, Upper Saddle River, NJ, USA, 1999.

[MD05] Tom Marrs and Scott Davis. *JBoss at Work: A Practical Guide*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2005.

[MEW08] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *LINQ in Action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[Mic09] Sun Microsystems. JSR 317: Java Persistence 2.0. `http://jcp.org/en/jsr/detail?id=317`, March 2009.

[Mit03] John Mitchell. *Concepts in Programming Languages*. Cambridge University Press, Cambridge, United Kingdom, 2003.

[Möl09] Ralf Möller. Lecture Notes on "Einführung in Datenbanksysteme". `http://www.sts.tu-harburg.de/~r.f.moeller/lectures/db-ws-08-09.html`, January 2009.

[NET06] The .NET Standard Query Operators. Technical report, Microsoft Corporation, Redmond, WA, USA, 2006.

[OO00] Patrick O'Neil and Elizabeth O'Neil. *Database-Principles, Programming and Performance*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, Raleigh, NC, USA, 2007.

[Par08] Doug Pardee. The Scala for comprehension from a Java perspective. `http://creativekarma.com/ee.php/weblog/comments/the_scala_for_comprehension_from_a_java_perspective/`, January 2008.

[Pol09] David Pollak. *Beginning Scala*. Apress, Berkely, CA, USA, 2009.

[PR07] Paolo Pialorsi and Marco Russo. *Introducing Microsoft®LINQ*. Microsoft Press, Redmond, WA, USA, 2007.

[Pri08] Rakesh Prithiviraj. IDE Customization to Support Language Embeddings. Master's thesis, Institute for Software Systems, Hamburg University of Technology, Hamburg, Germany, 2008.

[RT08] Karel Richta and David Toth. Formal Models of Object-Oriented Databases. In *Objekty 2008*, pages 204–217. Prague, Czech Republic, 2008.

[Sei04] Peter Seibel. *Practical Common Lisp*. Apress, Berkely, CA, USA, 2004.

[Sta07] Mike Stay. Category Theory for the Java Programmer. `http://reperiendi.wordpress.com/2007/11/03/category-theory-for-the-java-programmer/`, November 2007.

[Teu06] Jens Teubner. *Pathfinder:XQuery Compilation Techniques for Relational Database Targets*. PhD thesis, Lehrstuhl für Datenbanksysteme, Institut für Informatik, Technische Universität München, München, Germany, 2006.

[Tri09] Konstantin Triger. Java integrated Query (jaQue). `http://code.google.com/p/jaque/`, May 2009.

[Wei98] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Wen09] Kaichuan Wen. Translation of Java-Embedded Database Queries with a Prototype Implementation for LINQ. Technical report, Institute for Software Systems, Hamburg University of Technology, Hamburg, Germany, 2009.

[WPN06] Darrin Willis, David Pearce, and James Noble. Efficient Object Querying for Java. In *Proc. of the European Conf. on Object-Oriented Programming*, pages 28–49, Nantes, France, 2006.

[YM98] Clement T. Yu and Weiyi Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[ZHR+06] Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Tom Risser, and Mark Hoeber. *The Java Tutorial: A Short Course on the Basics, 4th Edition (Java Series)*. Prentice Hall, Upper Saddle River, NJ, USA, 2006.

Index