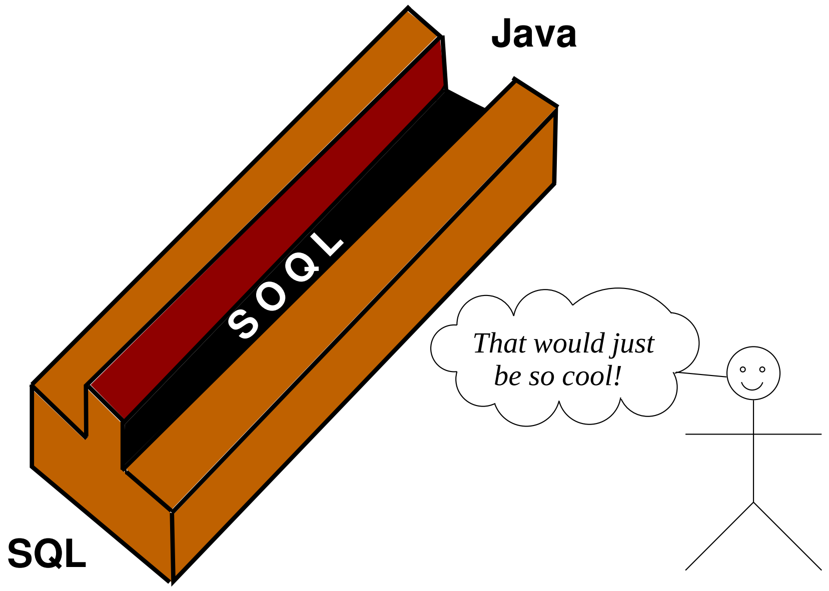
Simple Object Query Language in PersiJ



Expressing declarative SQL queries in object-oriented Java-like syntax

AUTHORS
Rolf Njor Jensen
Peter Sönder

Master Thesis
June 2006

Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:
Simple Object Query Language in PersiJ

SUBTITLE:
Expressing declarative SQL queries in object-oriented Java-like syntax

PROJECT PERIOD:
Dat6, Cis4,
Feb 1st - June 1st 2006

PROJECT GROUP:
d630a

GROUP MEMBERS:
Rolf Njor Jensen

Peter Sönder

SUPERVISOR:
Per Madsen

COPIES: 4
PAGES: 102

Abstract:

The impedance mismatch problem of using Relational Database Management Systems in statically typed object-oriented programming languages has been subject to many partial solutions, and has many facets. This project concerns the querying part of such a solution, and tries to explain deficiencies found in existing querying methods through analysis and a literature study.
The goal is to ease the querying process by designing a new querying language that integrates with the host language, and is statically checkable, has the same syntax and very similar semantics as the host language, while being modularizable and optimizable.
Through a description of the new language – Simple Object Query Language (SOQL) – and a description of how it may be translated to SQL, we argue that it fulfills most criteria. Subject to development of a compiler and surrounding framework, it could be a favorable alternative to existing solutions as it among other properties enables static checking while still being modularizable and optimizable. The language fails to retain the same semantics as Java, the differences being intricate, and in conclusion it is questionable whether SOQL makes it any easier to express queries.

At the time, Nixon was normalizing relations with China. I figured that if he could normalize relations, then so could I.

Ted Codd, father of the relational data model

Preface

Prerequisites

Prerequisites to this thesis is intermediate knowledge of Java 1.5, Structured Query Language (SQL), object-oriented programming and modeling, and the relational model.

Reading notes

During the course of this thesis, a considerable number of acronyms is introduced. The first time an acronym is presented, it is written in full, followed by the shorthand form in parenthesis: Three Letter Abbreviation (TLA). Subsequent uses of the acronym is the shorthand form. At the beginning of this thesis there is a complete list of the acronyms used in this thesis.

Typographical notes

In the rest of the thesis, text that is source code of some sort will be printed with this font: `System.out.println("Hello World!");` or using a figure as shown in Figure 1.

```
System.out.println("Hello World!");
```

Figure 1: Example of source code.

Contents

List of Acronyms

Atomicity, Consistency, Isolation, and Durability (ACID)

The ACID properties are used to describe the transaction properties of a database management system. By fulfilling these properties one can guarantee that the database always is in a consistent state. *8, 10*

Abstract Compiled Query (ACQ)

A class where each instance is used to collect the different parts of a query during compilation. The Abstract Compiled Query can at the end of compilation be used to produce the transformed query. *49*

Application Programming Interface (API)

An Application Programming Interface (API) is the interface that a computer system or application provides in order to allow requests for service to be made of it by other computer programs, and/or to allow data to be exchanged between the two. *5, 8, 13, 26, 40, 41, 44, 45*

Annotation Processing Tool (APT)

The Annotation Processing Tool (or APT) is a utility by Sun Microsystems for processing annotations in Java. It provides a read-only view of the program structure and source code. When APT runs it can produce new source files (also other files like configuration files etc.). After this it can compile the original and newly generated source files [41]. *21, 33, 47–49*

Call Level Interface (CLI)

Interface to a RDBMS that uses text strings containing queries in the SQL syntax to perform operations on the database. *3*

db4objects (db4o)

db4o is an open source object-oriented database. It supports both Java and C#. db4o has successfully implemented Native Queries [14] as their way of querying [18]. *8*

Database Management System (DBMS)

A collection of programs used to store, modify, and retrieve information from a database. *9*

Extended Backus Naur Form (EBNF)

All EBNF constructs can be expressed in plain Backus Naur Form (BNF) using extra productions. EBNF is more readable and succinct than BNF. *83*

Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) technology is the server-side component architecture for the Java 2 Platform, Enterprise Edition (J2EE) platform. EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology [22]. The latest version of EJB is version 3.0, which we will be referring to as EJB. *4*, *13*, *20–23*, *25*, *26*, *29*, *31*, *33*, *85*

EJB Query Language (EJB QL)

EJBQL is the query language used in an EJB project. It resembles SQL in many ways. *14*

Hibernate Query Language (HQL)

HQL is the object-oriented query language used by the Hibernate framework [24]. HQL returns objects instead of plain data (as is the case when only using SQL). It supports notions like inheritance, polymorphism and association *5*, *6*, *13*, *14*

Integrated Development Environment (IDE)

An IDE is a piece of software that assist developers in writing source code through syntax highlight, debuggers, build-in compilers, and other tools that help simplifying the development process. *32*

Java Compiler (JavaC)

The Java Compiler tool reads class and interface definitions (source files), written in the Java programming language, and compiles them into byte code class files [26], which can then be executed on a JVM. *32*, *33*, *47*, *49*

Java Compiler Compiler (JavaCC)

JavaCC is a popular parser generator. It uses a grammar specification and creates a Java program that can recognize the grammar [27]. *34–36*, *49*

Java DataBase Connectivity (JDBC)

JDBC is an API for Java that wraps CLI's for several different databases. JDBC is maintained by Sun Microsystems. *3*, *4*, *10*, *16*, *27*, *31*, *33*, *51*, *52*, *68*, *72*, *75*

Java Data Objects (JDO)

JDO provides an abstraction to the persistent layer in Java. It has been published under the JSR 12 and the newest version 2.0 under JSR 243 [28]. *4*, *13*

Java Data Objects Query Language (JDOQL)

JDOQL is the query language used by JDO [28]. JDOQL follows the syntax of Java where possible. *13*, *14*

Java Persistence Query Language (JPQL)

JPQL is a part of the new Java Persistence API, and defined is in defined in JSR 220 [31] *13*

Java Virtual Machine (JVM)

The Java Virtual Machine (or JVM) is a virtual machine, thus the name, that executes Java byte code. *7*, *40*

Open DataBase Connectivity (ODBC)

A standard for an API that wraps CLI's for several different databases, providing a common interface regardless of the underlying RDBMS. ODBC is a standard co-opted by Microsoft from the SQL Access Group consortium. *3*

Object-Relational Mapping (ORM)

Object-Relational Mapping is a technique to link the relational database with the object-oriented language. There exists many tools to handle this, amongst others are Hibernate [24] and JDO [28]. *4*, *13*, *21*, *23*, *29*

Plain Old Java Queries (POJQ)

POJQ is a framework for writing and executing database queries written in Java [8]. POJQ uses Native Queries [14] and bytecode analysis to translate the Java code into SQL. The initial implementation uses JDO 1.0 [28], but it should also be possible to use EJB [22]. *14*

Relational Database Management System (RDBMS)

Software systems that manage and store data organized by the relational data model. See Abraham Silberschatz *et al* [1] for more verbose explanation. *1*, *6*, *9*, *15*, *16*, *40*, *43*, *65*, *72*, *73*

Software Developers Kit (SDK)

SDK is an environment for building Java applications. It includes tools for developing and testing. *21*

Simple Object Database Access (SODA)

SODA is an object API to gain access to databases. The current specification focuses on queries with goals like type safety and programming language independence, and was developed in Java programming language. *8*

Simple Object Query Language (SOQL)

SOQL is the object-oriented query language provided by the PersiJ framework. *16*, *20*, *29*, *31*, *32*, *34*, *37*, *40–43*, *52*, *54*, *59*, *65–73*, *75*, *76*, *83–86*

Structured Query Language (SQL)

SQL is a widely used language to query RDBMS. Provides constructs for insert, update, and deleting information in a RDBMS. *v*, *3*, *5*, *6*, *10*, *13–17*, *31*, *33*, *34*, *38*, *40*, *42–45*, *47*, *49*, *51–55*, *57*, *62*, *65–69*, *71*, *72*, *75*, *76*, *85*, *86*

SQL-Java (SQLJ)

SQLJ is a specification for handling SQL in a Java application. Before compiling the SQLJ source code it needs to be translated into valid Java code [40]. *4*, *5*, *14*, *70*

Sun Microsystems (Sun)

Sun Microsystems is a vendor of computers, computer components, the Solaris operating system and the Java programming amongst others [41]. *7*

Three Letter Abbreviation (TLA)

A meta-acronym. As the name suggests, used to describe acronyms of length three. *v*

1 Introduction

1.1 Setting the scene

The broader problem underlying the work of this thesis is the problem of using a Relational Database Management System (RDBMS) in software written using statically typed object-oriented programming languages. Connecting the two proves to be quite hard due to differences in data abstraction models, concurrency models, programming paradigms, primitive type definitions, etc. The problem is in general referred to with the term “Impedance Mismatch”, first introduced by Copeland and Maier [15] in a publication about adding persistence support to the language Smalltalk-80. Originally the use of impedance mismatch stems from electrical engineering, expressing how hard it is to interconnect two systems.

Throughout the last two decades, much effort has gone into solving the impedance mismatch problem, and while many solutions are widely adopted by industry, all solutions suffer from some deficiencies.

1.2 Report outline

This report is laid out in the following chapters:

Preliminary Analysis In the following chapter we investigate different solutions to the Impedance Mismatch problem, noting how each of them is deficient in some manner. Thereafter we consider why the premises are not changed to avoid the problem altogether. Then we use previous work to list a number of criteria that a solution to the impedance mismatch problem must fulfill to properly solve the problem. Finally we reduce the scope of the project to only concern one facet of the problem.

Analysis The chosen facet is querying, and through an analysis of existing querying methods we find a number of criteria that a querying method must fulfill. Some basic choices for designing a new query language are made, and finally a problem statement concisely puts forward the specific problem that is the subject of this project.

The PersiJ framework To be able to develop the new query language, a surrounding context must be available. PersiJ is the name of the persistence framework that is presented. PersiJ is largely based on existing persistence framework specifications, but modified to accommodate special requirements of the new querying language.

Simple Object Query Language Bit by bit, the new query language is designed, and this chapter gives an overview and informal description of the different parts of the language.

Transformation of SOQL Compiling the language is not a trivial task, and this chapter gives an informal description of the semantics, noting prerequisites for legal translations of the different parts of the language.

Discussion This chapter evaluates whether the language fulfills each of the criteria set forward in the analysis, and discusses to some degree the expressiveness of the new language compared to SQL.

Conclusion Finally, the conclusion summarizes the discussion and the major advantages and disadvantages of the new language are presented. On a last note, the conclusion is set in a wider context, taking a broader perspective on the results of the project.

Enjoy your reading.

2 Preliminary Analysis

This chapter starts by investigating existing (classes of) solutions to the Impedance Mismatch problem. Noting that they are all deficient in different ways, we pose the question: Why not change the premises? Arguing that relational databases cannot be replaced by object-oriented databases, the chapter proceeds by listing criteria that any solution to the impedance mismatch problem must all fulfill. Last we acknowledge that the scope of this project does not allow for the development of a solution to the whole impedance mismatch problem.
This chapter is largely based on our previous work [35].

2.1 Existing solutions

Currently, there are many solutions to the impedance mismatch problem, but unfortunately they all have one thing in common: they only solve a part of the problem. In the following, we present some of these solutions.

2.1.1 Call level interface

Call Level Interface (CLI) is an approach where it is possible to embed SQL statements as string variables in the source code which can be shipped to the database engine. The database then parses and executes the query and the result is returned. While CLI enables the programmer to use all of the features of the database, it is also subject to script injection problems [33] and lack of static checking. Examples of CLI are Java DataBase Connectivity (JDBC) and Open DataBase Connectivity (ODBC). Figure 2.1 shows an example of Java code using JDBC. The example is split in three parts. The first creates a connection to the database, the second performs a query on the database, and the third inserts a new person into the database.

2.1.2 Embedded SQL

Another approach is Embedded SQL, where SQL statements are embedded within the source code, and using a pre-compiler, the statements are transformed into CLI code. This gives some degree of static type checking, and the dissimilarity of primitive types are also handled. An example of embedded SQL is SQL-Java

```
// Initializing a connection to the RDBMS
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/somedb",
    "user",
    "password");

// Performing a query on the database
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT * FROM person WHERE fname = 'Peter'");

while (rs.next()) {
    String fname = rs.getString("fname");
    ...
}

// Inserting a tuple in the database
String fname = "Per";
Statement stmt = con.createStatement();
stmt.executeQuery(
    "INSERT INTO person (id, fname, lname) " +
        "VALUES ('', '" + fname + "', 'Madsen');"
);
```

Figure 2.1: Example of JDBC code.

(SQLJ) [40]. Figure 2.2 shows an example of SQLJ. As with the JDBC example in Figure 2.1, we have three parts. The first creates the connection to the database, the second queries the database, and the third inserts a new person into the database.

2.1.3 Object-Relational Mappers

After the initiation of the development of the object-oriented programming paradigm, a lot of work began regarding providing persistent storage support for objects. Carey and DeWitt [10] gives an overview of the development effort from the mid-80's until the mid 90's, identifying trends, and making predictions as to the future development trends. One of the key areas of future development identified by Carey and DeWitt in 1996 was "Object-Oriented Client Wrappers" - *the use of object wrappers to support the development of object-oriented client-side applications against legacy databases* [10]. These wrappers have since evolved into Object-Relational Mapping (ORM) frameworks, and are in widespread use. Solutions in this category includes Hibernate [24], TopLink [43], Java Data Objects (JDO), Enterprise JavaBeans (EJB), etc.

Deficiencies common to these solutions is the need for metadata to identify

```
// Initializing a connection to the RDBMS
Oracle.connect(
    "jdbc:oracle:thin:@localhost:1521:somedb",
    "user",
    "password"
);

// Performing a query on the database
SelRowIter result = null;
String fname = "Peter";
#sql result = {
    SELECT * FROM person WHERE fname = :fname
};

while(result.next()) {
    fname = result.fname();
    ...
}

// Inserting a tuple in the database
String fname = "Per";
#sql {
    INSERT INTO person (fname, lname) VALUES (:fname, "Madsen")
};
```

Figure 2.2: Example of SQLJ code.

and describe mapping properties, complication of build process, discrepancies of primitive data types, etc. One of the biggest problems however, is the lack of static type checking when using string based query approaches – or very complicated and verbose query expressions [35].

Figure 2.3 shows an example of Java code utilizing Hibernate and its Hibernate Query Language (HQL). The code presupposes that the `Person` class has been annotated with metadata, and that the rest of the Hibernate framework is properly configured with database connection information, etc.

2.1.4 Multi-paradigm languages

Recently Microsoft released C# 3.0 (from this point on referred to as C# [16]). Included in C# is the persistence features of research language Cω [12] [7]. The object-oriented model of C# has been extended with anonymous types, traits, expression types, etc., to enable a large syntactic addition to the language that is somewhat like SQL. The now-included query language can be used both to query object structures as well as external data sources for which some Application Programming Interface (API) has been implemented - including relational data sources.

```
// Set up the environment
Session session = HibernateUtil.currentSession();
Transaction tx = session.beginTransaction();

// Update the persistent representation of an object
Person person = new Person();
person.setFName("Per");
person.setLName("Madsen");

session.save(person);
tx.commit();

// Perform a query on the database using HQL
Query query = session.createQuery(
    "FROM Person AS p WHERE p.fname = :fname");
query.setCharacter("fname", 'Per');

// Handle the query result here
for (Iterator it = query.iterate(); it.hasNext()) {
    Person p = (Person) it.next();
    ...
}

HibernateUtil.closeSession();
```

Figure 2.3: Example of Hibernate using HQL.

The language is extended with one more paradigm (declarative for querying), and becomes a larger language - potentially harder for the system developer to master. Figure 2.4 shows an example of querying in C# - and as it can be seen, the query language is SQL-like, and integrated as a part of the language. In the example, a new anonymous type is built from the query, which in fact is just a type that has not been given a type (the anonymous is what makes the querying possible in C# as the return type is not known before the query is executed). An alternative is to just select the objects directly - and populate new instantiations of an existing type.

2.2 Why not change the premises?

Since it is a problem to use the statically typed object-oriented languages in conjunction with RDBMSs, why not change the premises and use another persistent storage mechanism?

```
// Establish a query context over ADO.NET connection
DataContext context = new DataContext(
    "Initial Catalog=petdb;Integrated Security=sspi");

// Grab variables that represent the remote tables that correspond to
// Person and Order CLR types
Table<Person> custs = context.GetTable<Person>();
Table<Order> orders = context.GetTable<Order>();

// Build the query (using a SQL-like syntax)
var query = from c in custs, o in orders
    where o.Customer == c.Name
    select new {c.Name, o.OrderID,
                o.Amount, c.Age};

// Execute the query and print the result
foreach (var item in query)
    Console.WriteLine("{0} {1} {2} {3}",
            item.Name, item.OrderID,
            item.Amount, item.Age);
```

Figure 2.4: Example of C# code - taken from [32].

2.2.1 Orthogonal persistence

One approach is to use orthogonally persistent object-oriented languages. Orthogonal persistence is described by Malcolm P. Atkinson [4], who is one of the creators of PJama [3]. The PJama project aimed at implementing orthogonal persistence in Java, but unfortunately the development was stopped because their choice of Java Virtual Machine (JVM) was abandoned by Sun Microsystems (Sun). In [4] Malcolm P. Atkinson uses three properties that must apply for orthogonal persistence to be fulfilled:

- *Orthogonality* – The persistence facilities must be available for all data, irrespective of their type, class, size or any other property.
- *Completeness or Transitivity* – If some data structure is preserved, then everything that is needed to use that data correctly must be preserved with it, for the same lifetime.
- *Persistence Independence* – The source and byte codes should not require any changes to operate on long-lived data. Furthermore, the semantics of the language must not change as the result of using persistence.

There exists several examples of orthogonal persistence applied to the object-oriented paradigm. One of the first was the programming language Self [44], whose runtime (and development) environment saves the complete state of the application in a “snapshot”.

Although PJama was abandoned, there was a number of lessons to be learned. Orthogonal persistence might seem as an attractive alternative at a first glance, but there are several factors that lead to a disfavoring of this approach. In an orthogonally persistent system, it is the state of the *whole* system that is persisted, and upon restoration, it is the *whole* state being restored – all classes, objects, attributes, GUIs etc. An implication of this is the lack of ability to persist and restore only subsets of a total storage.

Sharing persisted state between concurrent executing applications becomes difficult, moreover, the Atomicity, Consistency, Isolation, and Durability (ACID) properties of transactions could not easily be applied to the Java platform [5], and therefore they had to investigate how a new transaction model could be applied.

2.2.2 Object-oriented databases

One of the predictions of Carey and DeWitt [10] was the demise of object-oriented databases. This has shown to be partly true, with object-oriented databases only gaining acceptance as niche products [30]. While removing some of the impedance mismatch, existing object-oriented databases do however, still suffer from some deficiencies. For one thing, expressing queries is still based on declarative queries without static type checking, or depend upon the construction of programmatic models representing queries.

Figure 2.5 shows an example of a programmatic model representing a query. One example of a current object-oriented database is db4objects (db4o). db4o has one very interesting property: while still providing programmatic query models and string based queries, the newest release has implemented Native Queries [14].

```
// An example of the Simple Object Database Access (SODA) API in db4o
Query query=db.query();
query.constrain(Pilot.class);
query.descend("name").constrain("Per Madsen").not();
ObjectSet result=query.execute();
```

Figure 2.5: Example of the SODA query API in action in db4o.

Native Queries is an approach presented by William R. Cook and Carl Rosenberger [14] that uses the syntax of the host language (in the case of db4o it is Java and C#), and by pre-compiling certain methods designated as queries and performing byte-code analysis on the referenced methods, the object-oriented code is translated to database calls. We will be going into details about Native Queries in Section 3.1.3.

2.2.3 Sticking to relational databases

There are a number of reasons to stick to relational databases and deal with the impedance mismatch, rather than changing to another form of persistent storage.

Legacy data: The relational data model was first introduced in 1970 by C.F. Codd [11], and has been subject to intensive development since. Database Management System (DBMS) have been around for nearly as long, and as a consequence hereof, many applications with the need for some sort of persistent storage is going to operate on legacy data that is not easily migrated to another form of persistent storage.

Theoretical foundation: Due to extensive research efforts during the past three decades, the relational data model, and relational databases rest upon a solid and extensive theoretical foundation - enabling the DBMS to have grown to become complex and highly efficient.

Maturity: The RDBMSs that are in widespread use (e.g. SQL Server [34], Oracle Database [36], DB2 [17], PostgreSQL [37], etc.) have meta facilities (including backup facilities, platform utilization, etc.) and are quite mature, as they have been under development and in production environments during the past three decades.

Partial persistence: Unlike some systems that are orthogonally persistent, a relational database provides the opportunity for partial persistence where only part of the application's state gets persisted. More relevant perhaps, is also that the application does not have to load all of the persisted state upon application startup. This is relevant in scenarios where the amount of persisted data is simply to much to handle in-memory.

2.3 What's the problem

While there are many suggestions to solutions to the impedance mismatch problem, Cook and Ibrahim [13] have tried to identify the criteria that a solution must fulfill in order to solve the impedance mismatch problem, and which properties can be used to describe different solutions. One of the conclusions of the article is that for one solution to properly solve the problem, all of the criteria must be fulfilled.

In our previous work [35] we evaluated these criteria, and while we where reviewing some existing solutions we ended up extending the set of available criteria. Table 2.1 lists an overview of these criteria. Those marked with Δ were introduced in our previous work. Those marked with $\otimes$ are added during this project.

Criteria	**Description**
Static checking	Having static checking is a huge advantage for the system developer. Static checking enables the ability to check code at compile time, type as well as semantics. This greatly decreases the risk of encountering run time errors.
Interface style	Interacting with the database either through SQL or the host language is something that need to be considered. Some existing solutions to the impedance mismatch problem breaks with the object-oriented paradigm thus ending up as a multi paradigm language. The level of persistence should also be taking into account.
Type mismatch issue	Either the problem of matching the type system from the programming language to the database is handled directly by the system developer (i.e. in JDBC) or it is handled by the framework or language extension (i.e. with Hibernate).
Reuse	To what extent it is possible to reuse parts of the query in different contexts. In other words - is it possible to modularize queries, and combine them at compile- and runtime?
Concurrency	How does the application support concurrency and how does this concurrency map to the ACID properties? Should the model for concurrency be changed?
Optimization	Whether or not the system developer should be able to do optimization. This could be through criteria shipping, grouping queries, or prefetching related objects.
Build process Δ	Depending on whether a framework or language extension is the preferred choice, there might be changes to the build process. A language extension would require a pre-compiler, if the extension is not to be build into the language itself.

continued on next page

continued from previous page

Tool support Δ	To what extent there is tools that support the framework or language extension.
Language alteration Δ	How many, if any, changes are made to the host language.
Schema evolution Δ	How is changes in the database schema handled by the framework or language extension.
Partial persistence $\otimes$	Does the solution allow for persisting only a subset of the application state, and does it allow for instantiating only a subset of the persisted state?

Table 2.1: Criteria that can be used for evaluation.

William R. Cook and Carl Rosenberger conclude:

> *A complete solution to the problem of impedance mismatch must provide both a clean programming model and high performance. While issues of mapping data between databases and programming languages have largely been resolved, significant issues remain. The interface should leverage the best capabilities of both databases and programming languages to for optimization, static typing, and modular development. Each of these aspects has a solution by itself. The problem of impedance mismatch is meeting all the goals simultaneously. ...* [13]

In other words, they conclude that the solutions they have reviewed, all are deficient in some way, and no solution meets all criteria at once. The scope of this project does not allow for the development of a full solution to the impedance mismatch problem, and will instead focus on solving one particular facet of the problem.

Before proceeding with a description of the particular facet, there are a few other areas that we would like to remove from our focus. One of these is concurrency. Although concurrency is an important aspect when working with databases, we believe that adding the aspect of concurrency to this project would result in an extra layer of complexity thus removing our focus from the facet that we find intriguing. We will also remove our focus from schema evolution. We will be touching the subject of schema evolution without providing an actual solution, because this as concurrency lies outside our choice of facet.

3 Analysis

In this chapter we will use the preliminary analysis as an outset to choose a facet of the impedance mismatch problem. We will then analyze this facet with respect to existing solutions, and describe a set of criteria by which a solution to the facet can be evaluated.

We proceed by presenting the method we will follow to answer the question of whether it is possible to solve the facet with respect to the presented criteria, and end with a problem statement that concisely states the problem we will try to solve.

3.1 Querying approaches

The facet that will be treated in this project is the fashion of querying that a solution to the impedance mismatch problem provides. All the existing kinds of solutions have deficient ways of performing queries. The following will treat each kind of solution, noting how the querying method is deficient. The end of this section will summarize the characteristics that can be used to describe a querying fashion, and present a set of criteria that we wish to fulfill with respect to a new way of querying.

3.1.1 Object-Relational Mappers

ORM solutions such as Hibernate, JDO, EJB, etc. provide good facilities for mapping relations to objects, balancing the need for fine-grained programmer control and ease-of-use. One of the problems, namely the separation of mapping metadata from the source code can be addressed with Java 1.5's annotations and the latest EJB specification. Unfortunately, the querying mechanisms of the ORM tools still lack elegance. They provide special object-oriented querying languages (e.g. HQL, Java Persistence Query Language (JPQL), and Java Data Objects Query Language (JDOQL)), but static type checking is lacking, and while partly operating on objects, the querying language is still declarative. Alternatively the frameworks provide support for SQL queries and programmatic querying, using an API that although native to the host language, is somewhat cumbersome to use.

3.1.2 Embedded queries and multi-paradigm languages

SQLJ and C# amend the language syntax, and allow some sort of SQL inline in the host language. Both solutions solve the static check issues. Although C# integrates some sort of SQL even deeper, enabling querying on object structures, both solutions are deficient in one very important way. In an object-oriented language they introduce a new paradigm - a declarative query language. Our belief is that this enlarges and complicates the language unnecessarily.

3.1.3 Native Queries

The previously mentioned Native Queries [14] approach has spurred a project named Plain Old Java Queries (POJQ) on `java.net`, which aims to provide support for Native Queries for JDOQL, but HQL and EJB Query Language (EJB QL) could also be used with POJQ.

The concept of Native Queries is however quite interesting, since it inherently solves that static type checking problems. Moreover, since it operates with the syntax of the host language, and similar semantics, it does not require the system developer to master several paradigms.

Figure 3.1 shows a simple example of Native Queries that retrieves all the students that are younger than 20 and have an "A" grade.

```
// A Native Query - specifying the selection predicate by implementing
// the abstract method match in the abstract Predicate class

List students = database.query(
    new Predicate<Student>(){
        public boolean match(Student student){
            return student.getAge() < 20
                    && student.getGrade().equals(gradeA);
        }
    }
);

// The abstract Predicate class
abstract class Predicate <ExtentType> {
    public <ExtentType> Predicate () {}
    public abstract boolean match (ExtentType candidate);
}
```

Figure 3.1: Example of Native Queries in Java.

Although Native Queries give an elegant fashion of querying - i.e. that it uses the semantics of a language that the system developer is already familiar with to express queries, it does have one very important deficiency (in Java). In absence of anonymous methods (closures), the vehicle chosen to express the predicate (contained in the match method) becomes an anonymous class that implements the abstract class `Predicate`. This design choice unfortunately gives a quite verbose syntax when specifying queries, and is specific to Java.

While the verbosity is a significant drawback, the underlying idea of Native Queries, namely the ability to express queries in a syntax and semantics native to the host language, is very appealing.

3.2 Querying method criteria

Considering these query approaches, we propose the following criteria to use when evaluating a new querying approach:

Static checking: The query form must enable compile time static type and semantic checking. This is one of the biggest deficiencies of string-based queries, and is necessary to overcome.

Automatic marshalling and unmarshalling: The conversion from the result of a query to the RDBMS which is a relation with tuples, into a collection of objects, must be performed by the query mechanism. At least it must be possible for the rest of the persistence framework to perform the marshalling and unmarshalling between the two data models.

Same paradigm as host language: The language used to express the query must be in the same paradigm as the host language. If the host language is object-oriented, then the query language must be so too. Moreover, the semantics of the query language and the host language must be as similar as possible, effectively making the task of writing queries no different than expressing anything else in the host language, and thereby make the querying process transparent.

Minimal verbosity: The typing required to represent a query must be minimal. If the querying becomes to complex and difficult to master one might as well be using SQL.

Minimal language alteration: The alteration of the host language needed to accommodate the queries, must be kept at a minimal level. This is to prevent the host language from growing unnecessarily.

Modularization: It must be possible to divide a query into different modules that can be combined or used individually.

Optimization: The query must be transformed into a query language that is native to the RDBMS – typically SQL. This way the RDBMS is at liberty to perform query optimization. Moreover, it should be possible to combine query blocks while retaining the possibility of letting the RDBMS optimize the query.

3.3 Choosing a language

When designing a new querying approach, it is necessary to decide which language to use as a starting point. We choose to base our project on Java. C# is another viable candidate, and the choice of Java is one based upon personal experience and knowledge to standard libraries, language syntax and semantics, etc.

3.4 Transformation

The source of the transformation will be syntactical correct Java 1.5 code that is wrapped in some language construct that designates it as a query. The target will be Java 1.5 code which utilizes JDBC to express the query in SQL.

3.5 Method description

In order to be able to answer our problem statement, we proceed by designing a query language. To support the language design process, a prototype of a compiler is built alongside with the design process. The development of the compiler is undertaken for the sake of exploring problematic areas, and it is not an explicit goal to build a working compiler for the whole language.

The language that the compiler is to transform, is to be some subset of Java (ideally all of Java) into other Java code. The development will start with a minimal language, which operates only on the basics of the Java language - namely classes, objects, references, primitives, messages, and branching. The question of whether it is possible to make a meaningful transformation of this minimal object-oriented query language (which we denote Simple Object Query Language (SOQL)) will be the first target. The design of SOQL is subject to the criteria listed in Section 3.2.

To make a meaningful compiler, the rest of the persistence framework must be in place. The persistence framework (which we denote PersiJ) is designed not to honor as many criteria as possible from Table 2.1, but merely to accommodate the needs from SOQL.

3.6 Problem statement

With this project we wish to answer the following question:

Is it possible to design an object-oriented query language integrated with Java, which can be transformed to Java code that ships the query as SQL to the database while still fulfilling the criteria: static checkable, automatic marshalling and unmarshalling, minimal verbosity, minimal language alteration, modularizable, and optimizable?

4 The PersiJ framework

In this chapter we present the PersiJ framework. We proceed by choosing to base the design of PersiJ on an existing persistence framework specification, and write how so. Thereafter a class (EntityCollection) is introduced as a part of PersiJ. Figure 4.1 shows how PersiJ is situated within an application.

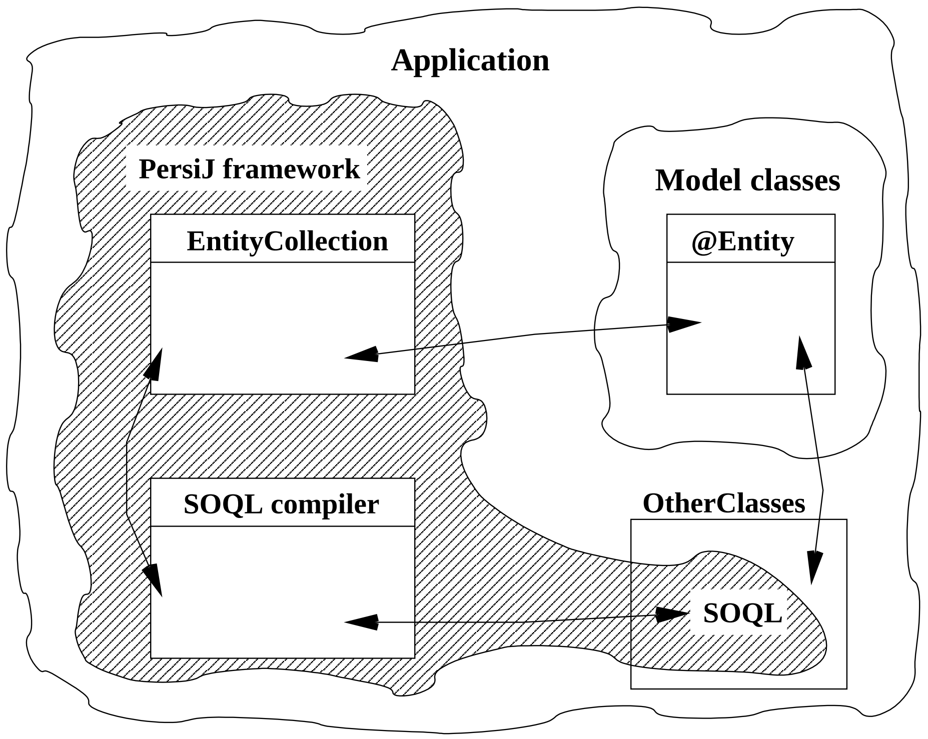


Figure 4.1: Illustrating the PersiJ framework within an application.

4.1 Using an existing persistence framework specification

Since building a complete persistence framework is not the focal point of this project, we base large parts of the PersiJ framework upon existing solutions - not for the sake of compliance with existing standards, but for the sake of “getting the job done” quickly.

We base some of the framework upon the proposed final draft of the EJB standard [22]. Most of the EJB standard is useful for us, as it describes in detail how a persistence framework could be. However, we have chosen to follow it only in part as we would like to be able to show off some particular language design properties of SOQL. The EJB describes mainly three parts - how entities work and are described (objects that are mapped to the database), how entities are managed (i.e. loaded from database and stored) and how they are queried. The two latter parts are not adopted in the PersiJ framework at all.

Much of the metadata that is required in such a persistence framework is in PersiJ present as annotations, and in the following section we present a discussion about this.

4.2 Identifying persistable types

Since we want to build a framework that is not orthogonally persistent, there needs to be some way to distinguish the classes whose instances have the ability to be persisted from those that do not. In the following we sketch some principal approaches for doing so, and acknowledge a few major advantages and disadvantages.

Inheritance: The inheritance hierarchy could be used to mark persistable classes. All persistable classes would then inherit from a common superclass. This makes it possible to utilize polymorphism and standard runtime reflection to distinguish between persistable and non-persistable classes. Unfortunately (in this case) Java only has single-inheritance, and requiring all persistable classes to inherit from one common class limits the expressiveness.

Marker interface: While interfaces normally express a number of abstract methods that implementing classes (or subclasses hereof) are required to implement or mark abstract, another alternative use of interfaces is as a *marker interface* [25]. A marker interface has no methods, and is used to indicate that an implementing class has some special property. We disfavor this approach, since we feel that an interface should indicate that an object has some special behavior. Using an interface to mark a property of an object is not indicating guaranteed behavior, but another property.

New class modifier: Another alternative is to modify the existing programming language, and introduce a new class modifier. Introducing new syntax into the language integrates the identification into the language. A disadvantage is however that the language grows, and gets more complicated. Moreover, a new (pre)compiler is required for the whole language.

External files: Properties files and XML files (with custom schema) have grown quite popular to hold configuration options and metadata - e.g. in conjunction with ORM. External files have two major disadvantages. First of all, the information they present is not integrated with the type system and secondly, the information is not located directly in the source code, next to the language elements that they concern.

Annotations: Since Java 1.5, annotations have been a part of the language type system. Annotations alleviate both of the major disadvantages of properties files and XML files - they place the metadata right next to the effected language element, instead of placing it in another location. Annotations are also integrated into the type system, and can be available through runtime reflection. The Annotation Processing Tool (APT) is distributed with the Software Developers Kit (SDK) for Java 1.5, and provides a framework for processing annotations.

In the PersiJ framework we have chosen to use *annotations* to enhance the source code. This way we can easily use APT to find the classes marked as entities.

4.3 Entities

The EJB standard describes an entity class as a class that is mapped to a database, or more precisely:

Definition 1 (Entity) *An entity is a lightweight persistent domain object.*

EJB proposes to use annotations as metadata. This leads us to adopt the entity class definition – to a certain degree.

The EJB specification treats how to identify entity classes, how to identify shadow information[1], how to distinguish between property based and member based access, which properties that entity objects have to uphold, how to map from entities to relations, how to map foreign key relations in the database to entities, etc.

The following is a specification of the entity class. Some of the differences compared to the EJB specification are small and intricate, and some are more massive. We have chosen to present the specification in some detail, to give a coherent view of the requirements to an entity class. Therefore, some of the succeeding text is equal to that of the EJB standard. We will explicitly note where the PersiJ entities differ from the EJB entities in more substantial matters. In general, where it has been possible, complexity and nice-to-have features have been removed from the EJB standard.

[1]Extra information added to the object in order to track it back to the tuple in the database.

4.3.1 Requirements on the Entity Class

The entity class must be annotated with the `@Entity` annotation. The entity class must be a top-level class, and have a public no-args constructor. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.

The persistent state of an entity is represented by instance variables, which may correspond to JavaBeans properties (see [22] for more information). Instance variables must not be accessed by clients of the entity. The state of the entity is available to clients only through the entity's accessor methods (getter/setter methods) or other business methods. Instance variables must be private, protected, or package visibility.

4.3.2 Persistent fields and properties

The persistent state of an entity is stored in its instance variables. An instance variable can hold persistent state if it is neither marked with the `@Transient` annotation, nor the `transient` modifier.

Mapping annotations are either applied to a field or its corresponding accessor method. The behavior is unspecified if both is the case. Mapping annotations on transient fields have no effect.

It is required that the entity class follow the method conventions for a JavaBean when persistent properties are used. In this case, for every persistent property called *property* of the entity, there is a getter method, `get`*`Property`*, and setter method `set`*`Property`*. For boolean properties, `is`*`Property`* is an alternative name for the getter method.

In addition to returning and setting the persistent state of the instance, the property accessor methods may contain other business logic as well, for example, to perform validation. The persistence provider runtime executes this logic when a property based access is used:

> *Caution should be exercised in adding business logic to the accessor methods when property-based access is used. The order in which the persistence provider runtime calls these methods when loading or storing persistent state is not defined. Logic contained in such methods therefore cannot rely upon a specific invocation order.* [22]

The behavior of runtime exceptions in the PersiJ framework is – unlike EJB – undefined. The reason for this is that we want to start out with a minimal language, where elements like exceptions have been removed.

The following types are present in the PersiJ framework:

- Primitive types (`int`, `float`, `byte`, `short`, `char`, `boolean`, `double`, etc.)
- `String`

- Other serializable types and wrappers of the primitive types such as `Integer`, `Byte`, etc.
- Enums
- Entity types and lists of entity types (`EntityCollection`, etc.)

The metadata for ORM are among others the annotations `@Table`, `@Column`, `@Id`, `@SecondaryTable`, `@JoinColumn`, `@ManyToOne`, `@OneToOne`, etc. For a full listing and specification we refer to EJB specification in [22]. We will not describe these as they are described in detail in the specification.

4.3.3 Mapping objects to relations

The EJB specification does not prescribe how to map objects to the persistent storage, so we need to look at this:

> *This specification does not prescribe how the abstract persistence schema of an entity bean should be mapped to a relational (or other) schema of a persistent store, or define how such a mapping is described.* [22]

We will therefore present four different ways of modeling this problem - all based on some work by Scott W. Ambler [2]. We will briefly summarize the four approaches:

1. **Map hierarchy to a single relation** – each hierarchy corresponds to one relation, with tuples for all the object members.
2. **Map each concrete class to a relation** – each object becomes one tuple in one relation.
3. **Map each class to its own relation** – each class corresponds to a relation. Each object becomes one tuple in each relation from its class until the base class.
4. **Generic mapping** – a number of relations contain tuples that describe the system: the classes, their members, inheritance relations and attributes. Finally there is one relation containing tuples with all the values.

The generic mapping (4) does not scale very well and queries quickly becomes very ineffective compared with the other three solutions. This is therefore not considered to be a usable approach at all, thus rejected.

Map each hierarchy to a single relation (1) removes the possibility of using domain constraints such as `not null` etc. thus is not a viable solution.

Mapping each concrete class to a relation (2) and mapping each class to its own relation (3) are both good alternatives. Let us have a look at the difference between the two in order to make a choice. Figure 4.2 illustrates the two ideas. In the left hand side of the figure, <A> means that class A is declared `abstract` thus each class is mapped to its own relation. Another thing worth noticing is that on the right hand figure class B both contains the attribute from A and B thus making schema evolution harder to maintain as adding a new attribute to A should propagate to all children inheriting from A. On the other hand, the solution on the left hand side need to map each child to its parent in order to fulfill the inheritance - e.g. through an id.

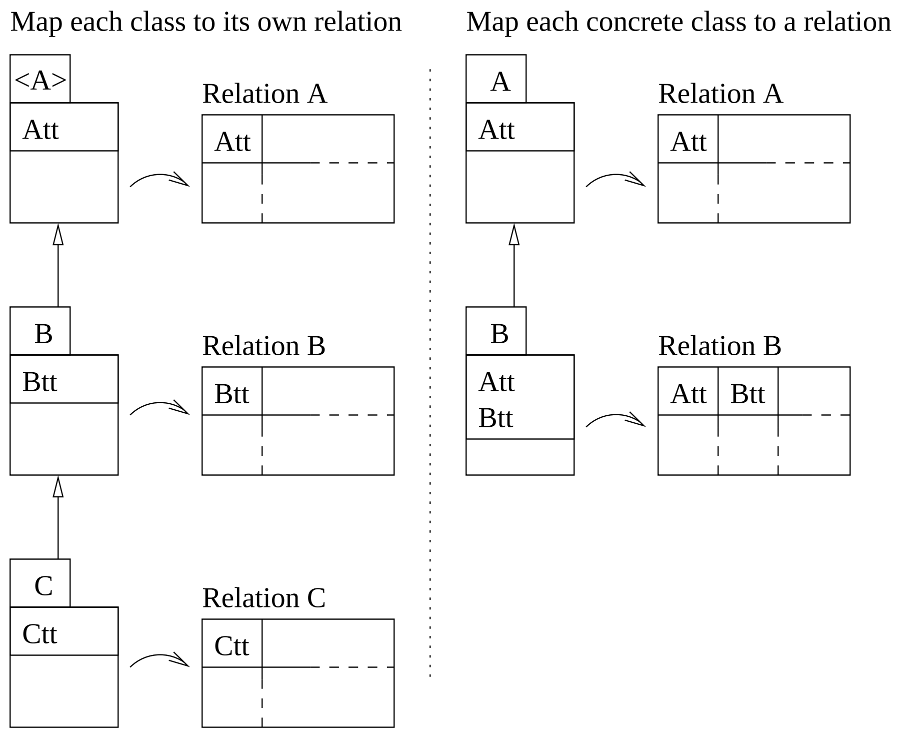


Figure 4.2: The two viable mapping approaches – (2) and (3).

Both solutions have some strong and weak points. Mapping each class to its own relation require that we model inheritance inside the database. Mapping each concrete class to a relation handles this by duplicating the inherited attributes on the cost of schema evolution. Because we are making a proof of case rather than having focus on an entire solution we are choosing *map each concrete class to a relation* (2) but recognizing that this solution might not be the best, if schema evolution will come in focus in future implementations. But for now, this will do as this is easier to implement.

4.3.4 Multi-valued properties

For multivalued properties – collections of objects – the entity must use the type `EntityCollection` (see Section 4.4). This is one area where PersiJ differs from

EJB. However, the metadata used to mark the relationships are equal, and so are their semantics. The following is from the EJB specification about entity relationships, but we have taken the liberty to change it to fit our layout.

> Relationships among entities may be one-to-one, one-to-many, many-to-one, or many-to-many.
>
> If there is an association between two entities, one of the following relationship modeling annotations must be applied to the corresponding persistent property or instance variable of the referencing entity: `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`.
>
> *These annotations mirror common practice in relational database schema modeling. The use of the relationship modeling annotations allows the object/relationship mapping of associations to the relational database schema to be fully defaulted, to provide an ease-of-development facility.*
>
> Relationships may be bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines the updates to the relationship in the database. The following rules apply to bidirectional relationships:
>
> - The inverse side of a bidirectional relationship must refer to its owning side by use of the `@mappedBy` element of the `@OneToOne`, `@OneToMany`, or `@ManyToMany` annotation. The `@mappedBy` element designates the property or field in the entity that is the owner of the relationship.
> - The many side of one-to-many / many-to-one bidirectional relationships must be the owning side, hence the `@mappedBy` element cannot be specified on the `@ManyToOne` annotation.
> - For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
> - For many-to-many bidirectional relationships either side may be the owning side.

For a full specification, we refer to EJB specification [22] once more.

4.4 The EntityCollection

Instead of using the `EntityManager` interface of EJB, we present a collection type native to the PersiJ framework - namely the `EntityCollection` (from this point written as EntityCollection to enhance readability). As we shall see later in Chapter 5, we want to be able to return a collection of entities from queries. Since it is a design goal to make the footprint of the framework as small as possible regarding API and language alteration, we combine the mapping of relations between objects, and the operations for manipulating entities into one type – the EntityCollection.

We proceed by describing the signature of the EntityCollection, and thereby explaining how it is supposed to work. All of the signatures can be seen in Appendix A.

4.4.1 Parametrized type

The EntityCollection is parametrized with one type. The parametrized type is the entity which the EntityCollection will contain, and the underlying collection type will contain. Constraining the type that the EntityCollection contains provides two major advantages:

- All the same advantages that are gained by making any other collection type parametrized apply - that the compiler statically can check validity of assignments and that explicit typecasts no longer is needed when using the collection.
- The EntityCollection can runtime infer the actual type parameter and from runtime reflection obtain the necessary information to perform mapping to the database.

4.4.2 Constructor

The constructor of the EntityCollection is overloaded. The default no-args constructor is used when the developer wishes to make a new, empty EntityCollection that can be used to persist entities.

The other constructor is a constructor used by the PersiJ framework internally. It takes a `java.sql.ResultSet` object as parameter, and iterates over the resultset - converting the rows from the result set to objects in the EntityCollection.

Figure 4.3 shows the two constructors in use.

4.4.3 Static data structures

Internally, the EntityCollection maintains data structures that map types and primary key values to references to corresponding loaded objects to assure that there is only one loaded copy of each entity.

```
// Creating and using a new blank EntityCollection
EntityCollection<Car> cars = new EntityCollection<Car>();
cars.add(new Car("Ferrari","Red","Expensive"));
cars.add(new Car("Koenigsegg","Yellow","Ridiculously expensive"));

// The other constructor used by the PersiJ framework
java.sql.PreparedStatement pStmt = con.prepareStatement(
   "SELECT * FROM cars WHERE pricetag = ?"
);
pStmt.setString(1,"Cheap");
java.sql.ResultSet resultSet = pStmt.query();
EntityCollection<Car> cheapCars = new EntityCollection<Car>(resultSet);
```

Figure 4.3: Example illustrating marshalling and unmarshalling.

4.4.4 JDBC connectivity

Statically the EntityCollection class maintains connections to the database. Should the system developer wish to interact with the database directly using JDBC, the method `getConnection()` can be used to retrieve a connection to the database. Figure 4.4 shows an example of `getConnection()`.

```
java.sql.Connection conn = EntityCollection.getConnection();
java.sql.PreparedStatement preparedStmt = conn.prepareStatement(
   "SELECT fname FROM person WHERE fname = 'Per'");
// Do more with the database/connection here
conn.close();
```

Figure 4.4: Example illustrating the method `getConnection()`.

4.4.5 Collection type methods

Since the EntityCollection implements the interface `java.util.Collection`, all of the methods from `Collection` are implemented. Internally the EntityCollection will contain some collection object which actually holds all the entities. The method calls implemented from the `Collection` interface are wrappers, that perform some form of action related to the persistence responsibilities, and re-delegate the call to the underlying collection implementation. In general, the wrapping functionality is responsible for keeping the internal static data structure up-to date. The persistent representation of the objects contained in the EntityCollection is not updated until the methods described in the following section are called.

One important issue is that the system developer needs to call `clear()` or `removeAll()` before deleting the last reference to the EntityCollection, to enable the static data structure to erase all references to the objects contained in the EntityCollection. If the EntityCollection is not cleared before being garbage collected, the static data structure will not become aware that the references to the objects that where contained in the collections now possibly are obsolete, and may be garbage collected [25].

4.4.6 Storing and deleting from persistent storage

Since the EntityCollection itself does not know whether or not an object is dirty (that is, in an inconsistent state with the persistent storage) there has to be a way of persisting the changes. There are four methods - two methods for persisting objects, and two methods for removing objects from persistent storage. Figure 4.5 shows an example of using the manipulation methods.

```
EntityCollection<Car> cars = new EntityCollection<Car>();
Car lamborghini = new Car("Lamborghini","Pink","Expensive");
Car mini = new Car("Morris Mini", "Racing Green", "Reasonable");
cars.add(lamborghini);
cars.add(mini);
// The collection cars now contain the lamborghini and mini - but they are
// not stored persistently.

cars.persist(lamborghini);
// The lamborghini is stored in the database.

lamborghini.setColor("Black");
// No more 'girly' colors.

cars.persist();
// The persistent representation of the lamborghini is updated
// and the mini is inserted.

cars.unPersist(mini);
// Delete only the mini from both the database and the collection.

cars.unPersist();
// ...and the lamborghini is deleted too.

lamborghini = null;
mini = null;
// ...and the objects can now be garbage collected by the JVM.
```

Figure 4.5: Example illustrating the `persist` methods.

4.4.7 Using prefecthing and lazy load

As all other ORM frameworks, PersiJ is subject to problems with loading whole object graphs upon retrieval of a single object. The EntityCollection allows for implementing some lazy load technique, where the related objects are loaded on-demand. As a first approach however, the EntityCollection blindly follows relations, loading the whole object graph. The PersiJ framework is subject to all the usual problems with lazy loading and prefetching, but using existing work on the subject [6], a satisfactory solution could probably be achieved.

4.5 Summary

The PersiJ framework can be considered as consisting of three parts:

Entities: Entities are objects which are mapped to the database. The metadata used to describe entities and relations between them is annotations, which quite closely follow the EJB standard.

EntityCollection: The EntityCollection is a collection that is used to manage entities, and relationships between them. Apart from the standard `Collection` methods, there are extra methods for storing and deleting objects.

SOQL: This is the query language used to retrieve objects from persistent storage. Look no longer than the following chapter for a description of SOQL.

5 Simple Object Query Language

In this chapter we will strive to design a method for querying within the PersiJ framework as laid out in the previous chapter. To summarize these: standard EJB annotations mark classes that are mapped to a database, and the EntityCollection type is used to model references between objects.

The modus operandi of this chapter is to start with deciding upon the basic form of the query construct, and hereafter finding a minimal subset of the Java language that can be used to perform queries. Constructs for expressing predicates, sorting, limiting and a way of modularizing queries is also presented. A number of examples are given to illustrate the ideas and concepts.

5.1 Basic form of query construct

In Section 3.1.3 we decided not to be using Native Queries as our starting point, due to the verbose fashion of implementing the `Predicate` class. The underlying idea of the `match` method (see Figure 3.1), namely the evaluation of a predicate on each candidate object, will also be the underlying idea for SOQL.

The queries have to marked in some way, so the compiler can recognize them. Following the fashion of EJB, SOQL queries are also marked with annotations.

Throughout the rest of this section, the form of the query construct is described.

5.1.1 Valid Java

One of the goals of the querying part of the PersiJ framework, is to investigate to which degree it is possible to express queries to relational databases using a language that resembles Java 1.5 as much as possible both with regards to syntax and semantics.

Whichever encapsulating language construct one might choose, the essential part of the compilation process will be the transformation from ordinary Java code to Java code utilizing JDBC and SQL statements.

To be able to focus the development process on this transformation, we want to be able to utilize a standard Java compiler (e.g. Java Compiler (JavaC)) to handle the compilation of the rest of the application. If this is possible, it is also possible to compile the source code and reflectively inspect the type system during the transformation of the query constructs.

This restriction enforces that we (re)use existing language constructs in Java, and that the queries that the system developer writes are actually syntactically and somewhat semantically sane (in the eyes of the Java compiler, anyway).

Sticking to standard Java 1.5 syntax also gives the benefit of tool support. Therefore, an existing Integrated Development Environment (IDE) like Eclipse [20] is able to operate on the source code with the query constructs.

Another approach would of course be to introduce new language constructs to mark up queries, but as we will see later, reusing existing language constructs enables a compilation (and transformation) process that simplifies the development of this compiler. On one hand, introducing new keywords or language constructs on method-level or class-level might better cover the semantics of querying, and make the code easier to read (and write). On the other hand, each new language construct complicates the language, and potentially makes it more difficult to learn.

We choose to use existing Java constructs in our design, because we believe that it is possible to use the host language (in this case Java) to express queries.

5.1.2 Encapsulating queries in methods

Given that we want to express the queries using existing Java language constructs, we have chosen to use a method as the vehicle for a query. There are two reasons for this. First of all, we do not feel that implementing anonymous classes like Native Queries gives an elegant solution. Another reason is that once the implementation takes place identifying these methods are somewhat easy - especially if they are marked in some specific way. We, on the other hand, acknowledge that methods are not the most elegant solution, but we feel that they are more elegant than anonymous classes.

We will be building the semantics for the query method little by little. Starting with the return type of a query, the result of some query is always a collection of objects that match the predicate expressed in the query. Using the EntityCollection as return type integrates SOQL with the rest of PersiJ.

The query method should be able to express a query, and return a collection of objects from the persistent storage that satisfies the predicate(s) expressed within the method. Given that we want to utilize a standard Java compiler, the language for these methods must be a subset to the Java language, and the signature of the method must be the same before and after the processing done by the PersiJ compiler.

The generated code that will constitute the compiled method might throw some exceptions - either from code supplied by the system developer, or from code interacting with the database, etc. Since exceptions are not a part of SOQL, we choose to move the exception handling outside the query method. Any exception thrown within the query method stemming from the database handling code is wrapped in a `PersiJException`, and re-thrown. The signature for the query method therefore includes `throws PersiJException`.

5.1.3 Using annotations for markup

For several reasons, we choose to distinguish the query methods from other methods by marking them with annotations. This is one of the places where we deviate from the EJB specification.

- We need to be able to distinguish query methods from other methods. That is, we need to identify methods working on persistent data in order to do the translation from Java source code to JDBC and SQL statements.
- Using annotations, we follow the style used to markup classes that are mapped to the database.
- Annotations is a native part of the Java language (since 1.5). This means that we can markup the language with annotations and simply use a standard Java compiler like JavaC.
- When using annotations to mark the methods, we keep the metadata close to the code, avoiding external configuration files.
- Instead of having to find the query methods ourselves, we can use APT to investigate the source base, and kick start the compilation process (more about this in Section 5.7).

5.1.4 Body of the query method

The method itself, and how the method can be called, is subject to normal Java semantics - this also means that normal access modifiers etc. apply.

The next task is to define how the method can be used to express a predicate that can be evaluated for each tuple in the database. Trying to solve this problem could be done several ways. When designing the language there are two design goals to keep in mind:

1. It should be easy to use (intuitive).

2. It should be compilable with any Java compiler (pure Java source code).

The subset of Java that we want to use is already informally defined, as we in Section 3.5 chose to only use the basics of the Java language - classes, objects, references, primitives, messages, and branching. This translates to taking away loop constructs, generics, exceptions, arrays, anonymous classes, labels, etc.

As a syntactical basis for our language we will be using the specification of Java 1.5 that can be found on the homepage for Java Compiler Compiler (JavaCC). This has then been reduced to remove the elements that we discarded in the analysis (exceptions, generics, etc.). After reducing this grammar - removing the above mentioned elements, the grammar for SOQL becomes *the base grammar*. The base grammar can be inspected in Appendix B.

The basic idea of the method (derived from Native Queries) is to have a reference to a candidate object in scope. Moreover, since the return type of the method is EntityCollection, a reference to such an object must also be in scope. If the candidate object is added to the EntityCollection, the candidate object must be contained within the returned collection. We therefore require that the method body starts with the two lines shown in Figure 5.1 (lines no. 3 and 4). Requiring them to be the two first statements is merely to ease implementation.

```
1 @Query
2 EntityCollection<type> someMethod(arguments) throws PersiJException {
3     EntityCollection<type> resultCollection = new EntityCollection<type>();
4     type candidateObject = null;
5
6     // The actual implementation
7
8     return resultCollection;
9 }
```

Figure 5.1: Example of the query method.

The arguments to the method, `arguments`, are optional. These can be input to the predicates, etc. The *type* in line no. 2-4 needs to be the same type as the type the EntityCollection returned from the method is parametrized with. To illustrate this, Figure 5.2 is a more usable example where the structure from Figure 5.1 is preserved. In Figure 5.2 we have replaced *type* with `Car`, the *resultCollection* with `cars` and *candidateObject* with `candidateCar`.

Now that we have covered the surrounding code, its time to look at the actual expression of a predicate itself. The simplest implementation of the method would only contain a add method that adds all elements to the EntityCollection. Figure 5.3 illustrates the actual implementation of Figure 5.2.

Instead of simply using the `add()` method (as described in Section 4.4) it is possible to use an if-statement to build up the predicate. Here we will briefly illustrate the idea, as the if-statement will be covered in greater detail in Section 6.6.

Figure 5.4 is an example of the if-statement. Here we are looking at cars and searching for all cars that are not *red*.

To ease the translation of Java source code to SQL statements, a return statement is mandatory, and must be the very last statement in the method block. This makes it easier to read and easier to translate. The return statements can be per-

```
@Query
EntityCollection<Car> getCars(arguments) throws PersiJException {
    EntityCollection<Car> cars = new EntityCollection<Car>();
    Car candidateCar = null;

    // The actual implementation

    return cars;
}
```

Figure 5.2: Example of the query method.

```
...
cars.add(candidateCar);
...
```

Figure 5.3: Without a predicate inside query method.

```
...
if(candidateCar.getColor().equals("red")) {
    // Do nothing
} else {
    cars.add(candidateCar);
}
...
```

Figure 5.4: With a predicate present inside query method.

ceived a bit like go-to statements (as they break out of the structure) [19]. As we will be discussing in Section 7.1.3 these if-statements are not to be evaluated as regular if-statements, and therefore a return statement inside these would be misleading.

5.1.5 The base grammar

Instead of simply showing examples, Let us review parts of the grammar from Appendix B. First, consider the block called PersijMethodBlock, which can be found in Table 5.1.

What is interesting here is the way we are constructing the query method. As mentioned earlier, the first two lines in the method body are predefined. In the grammar provided by JavaCC you would find the return statement inside the *Block-Statement* and not in the method block as we have chosen to do this. Again, because

```
PersijMethodBlock  ::=  {
                        EntityCollection <Type> <IDENTIFIER>
                              = new EntityCollection <Type> () ;
                        Type < IDENTIFIER>  = null ;
                        ( BlockStatement )*
                        return <IDENTIFIER> ;
                        }
```

Table 5.1: Grammar for *PersijMethodBlock*.

this could be misleading.

Another part of the grammar that is different from JavaCCs grammar is the *Statement* block illustrated in Table 5.2. This has been reduced drastically from the JavaCC version in order to remove excessive statements (for, while, switch, try, etc.). It is only possible to make predicates using the if-statements.

```
Statement  ::=  Block
                | StatementExpression ;
                | IfStatement
```

Table 5.2: Grammar for *Statement* block.

5.1.5.1 Types

The types explicitly present in the base grammar are `String` and the primitive types, `boolean`, `char` and `int`, and of course the EntityCollection (see Section 4.4). Apart from these, all qualified type names are allowed in expressions, etc.

5.1.5.2 Assignment operators

Assignment operators have been boiled down to only being =, and it is only possible to assign the predefined lines in Table 5.1. In other words, declaring variables inside the body of the method is not possible, which means that any variables to a query must enter the method as an argument, or already be in the scope containing the method (i.e. class variables and instance variables).

5.1.5.3 Comparison operators

The comparison operators are restricted to only be <, <=, >, >= and ==.

5.1.5.4 Boolean operators

The allowed boolean operators are: &&, ||, !

5.1.5.5 Branching

Branching is possible by using if-statements. The form of the if-statements is similar to that of (ordinary) Java.

5.1.5.6 Expressions

Expressions are equal to those of Java, and within both statements and expressions, it is possible to do both method calls and field dereferencing.

5.1.5.7 Method calls

Method calls are the basic mechanism of communicating in an object-oriented programming language, and have their natural place also in SOQL. Translating a method call on an object from SOQL to Java, is however not without difficulties. Due to the nature of the translation, it becomes important to be able to identify which methods alter the state of their objects, and which do not. We will elaborate in detail in Section 6.5 why this is so.

For now, consider Figure 5.5. In line no. 4, a safe method call is made on `paramColor`. It is safe for several reasons. The result of the call is not dependent on the candidate object – the `equals` method call does (apparently) not alter state of the object, nor any other objects.

```
 1 public EntityCollection<Car> getCarsByColor(Color paramColor, Counter cntr)
 2                                                  throws PersiJException {
 3     EntityCollection<Car> cars = new EntityCollection<Car>();
 4     Car candidateCar = null;
 5     if(paramColor.getName().equals(candidateCar.getColor().getName())) {
 6         cars.add(candidateCar);
 7     }
 8     candidateCar.getOwner().setName("Per");
 9     cntr.count();
10     cntr.infiniteRecursiveMethod();
11
12     return cars;
13 }
```

Figure 5.5: Using a safe and an unsafe method call.

An unsafe method call is made in line no. 7. Apparently the method call should be equivalent to renaming all cars owners to “Per”, but it is not safe since it alters the state of objects that are in the database. Another unsafe call is made in line no.

8. The `count` method on the object `cntr` presumably alters the state of the `cntr` object, increasing some internal variable.

When translating this method a problem arises - should the method be called once for each candidate object being evaluated? Should the method only be called once? Line no. 9 introduces yet another complication. If method calls are allowed, then so is recursion. Recursion per se is not problematic - however, infinite recursion is a problem. Non-termination can not be expressed in SQL and a classical result of computability is that the detection of non-termination is impossible - an instance of *"The halting problem"* [39].

Many of the problems with unsafe method calls come down to identifying whether a method alters state (has side-effect). This is a problem that has been investigated in some detail.

Dealing with side-effects Although this sounds like an easy task, it isn't. First of all, what is a side-effect? Is a method side-effect free if and only if it doesn't change the object in question? What happens if the method changes the state of another object - is it then without side-effects?

In the object-oriented programming language Eiffel [21] there is a strict distinction between a function and a procedure. A function does not call a procedure and does not change anything, but simply returns the value (is also called *pure*) whereas a procedure changes something. But in Java we cannot distinguish between functions and procedures. One way is to use annotations to markup pure methods (e.g. using `@pure` [29]), but this only solves part of the problem - since it then is up to the programmer to guarantee that the method only returns a value.

A lot of research has been done in the area of side-effects. Alexandru D. Salcianu and Martin C. Rinard [38] uses a technique called pointer analysis [45] to determine whether or not an object is pure. They distinguish between *read-only* parameters and *safe* parameters:

> *A parameter is read-only if the method does not mutate any object reachable from the parameter... A parameter is safe if it is read-only and the method does not create any new externally visible heap paths to objects reachable from the parameter.* [38]

There are three approaches to dealing with side-effects in an object-oriented environment. Either ignore side-effects, deny side-effects, or allow side-effects. The question then remains. Should we simply ignore side-effects and hope for the best? Or should we allow side-effects and try to resolve them, and if this fails inform the developer of the problem or should we simple deny side-effects once and for all. Denying side-effects completely would not be a viable solution, as we would need to define *what a side-effect is* - and be able to recognize this. Ignoring them could result in strange and unexpected behavior of the system. Therefore we must allow side-effects to be present in the system, and then find a way to resolve these.

Using an approach like Alexandru D. Salcianu and Martin C. Rinards are appealing, as analysis-based approaches does not require e.g. the `@pure` annotation to be added to the application.

5.2 Terminology

A few definitions are in place.

Definition 2 (Query Method) *A query method is a method in a standard Java 1.5 class definition that is marked with the annotation* `@Query`*. A query method must follow the grammar defined in Section B. The return type of a query method is the generic type EntityCollection.*

Definition 3 (Candidate Object) *A candidate object is an object of the same type as the type-argument to the return type of the query method. A candidate object must be of a type that is decorated with the* `@Entity` *annotation, and is a required part of a query method.*

Definition 4 (Persistable Object) *A persistable object is an object of a class that is marked with the* `@Entity` *annotation.*

All possible candidate objects (i.e. all objects of the type that are present in persistent storage) are evaluated with respect to the predicate expressed in the query method. If a candidate object fulfills the predicate, the candidate object is included in the set of returned objects.

5.3 Predicates

Expressing predicates in the most basic form is done using if-statements. Consider the following phrase: "*If a car is colored yellow, add it to the collection of returned objects.*". The predicate being tested for is of course whether the color of the car equals yellow - and expressing this in normal Java-like syntax is quite straightforward (see Figure 5.6).

```
...
if(candidateCar.getColor().equals("Yellow")) {
    cars.add(candidateCar);
}
...
```

Figure 5.6: Expressing a simple predicate.

To be able to translate the simple predicate, it is necessary to assign special semantics to the `equals` method, and assume that it can be translated to the equality comparison operator in SQL for primitives and strings.

Using both the "then" part and the "else" part of if-statements, and nesting if-statements, more complicated predicates can be expressed. This approach is pretty straightforward, and in Section 6.6 a detailed explanation of the translation to predicates is given.

If the expression in the if-statement does not contain any references to objects and primitives, the if-statement is not used to build a predicate, but is used to indicate actual control flow - branching. The branching can be used to dynamically build queries dependent on which branch of execution is followed runtime.

5.4 Sorting

The standard way of sorting a collection in Java, is by using the static method `sort`, and an anonymous implementation of a comparator. The sort method and a selected part of the `Comparator` interface is shown in Figure 5.7.

```
package java.util;
public class Collections {
    ...
    public static <T> void sort(List<T> list, Comparator<? super T> c);
    ...
}

public interface Comparator<T> {
    int compare(T o1,T o2);
}
```

Figure 5.7: Selected parts of `java.util.Collections` and `Comparator`.

Sorting a result from a query can be done likewise in SOQL. While it of course is possible to do this in the JVM, it can be moved as an operation that the RDBMS performs. To facilitate this, there are however some issues to solve.

5.4.1 EntityCollection implements List

The sort method accepts an object that is assignable to `java.util.List`, and EntityCollection therefore has to be altered to implement `java.util.List` instead of `java.util.Collection`. Changing the EntityCollection applies to all parts of PersiJ. The change will introduce a number of new methods in EntityCollection. The semantics of EntityCollection will also change a little, since a list according to the Java API is:

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. . . .

The shortcomings of the list (inefficient search complexity) may be overcome by the underlying implementation.

5.4.2 Which comparison operation

Letting the system developer provide a custom `Comparator` is unfortunately not possible. The solution is to provide two `Comparator` classes that are both part of the PersiJ API:

- `AscendingComparator`
- `DescendingComparator`

5.4.3 Which member to sort on

Since an object in an EntityCollection potentially has many members, it is not clear which member to sort on. Inherently, the `Comparator` interface does not provide any mechanism for indicating this - but since we want make the comparators work on all sorts of entities, this needs to be remedied. One solution would be to make a constructor on the comparators that takes a `java.lang.reflect.Member` object as parameter. The SOQL compiler could then use this to reflectively investigate which member the sort order should be determined after. However, obtaining this object for some candidate object does require some verbose coding for the system developer. Figure 5.8 shows an example of this.

Another approach that can coexist with the former, is to add a constructor to the comparators that simply takes an `Object object` as parameter. Upon creation, the comparator then receives the field that has to be sorted on as a parameter. In a normal runtime situation, the actual value of the parameter cannot be used to determine which field to sort on. But because the SOQL compiler inspects the source code, it can determine which member to sort on. This approach is illustrated in Figure 5.9.

The obvious drawback to this approach, is that primitives are not objects, and do not inherit from `Object` and can therefore not be used. In this case, the former approach may be used.

```
public EntityCollection<Car> getAllCarsSortedByColor() throws PersiJException {
    EntityCollection<Car> cars = new EntityCollection<Car>();
    Car candidateCar = null;
    cars.add(candidateCar);
    Collections.sort(
        cars, new AscendingComparator(
                        candidateCar.getClass().getField("color"))
        );
    return cars;
}

// Equivalent SQL statement
SELECT * FROM cars ORDER BY color ASC
```

Figure 5.8: Another example of the sorting mechanism in SOQL.

```
public EntityCollection<Car> getAllCarsSortedByColor() throws PersiJException {
    EntityCollection<Car> cars = new EntityCollection<Car>();
    Car candidateCar = null;
    cars.add(candidateCar);
    Collections.sort(cars, new AscendingComparator(candidateCar.getColor()));
    return cars;
}

// Equivalent SQL statement
SELECT * FROM cars ORDER BY color ASC
```

Figure 5.9: An example of the sorting mechanism in SOQL.

5.5 Limiting

Limiting the number of results is a feature of SQL that can be achieved in SOQL by adding a method to the EntityCollection that sets the maximum number of elements that it may contains. The method is given special semantics in the sense that if it is called in a query, it is translated to a SQL fragment.

Unfortunately there is no such method already present in the `Collection` interface, so we add one, and name it `setMazSize`. The reason that it is not called `setLimit` or similar, is to follow the existing concepts of the `Collection` interface, where the number of elements contained in a collection is refereed to as size. Figure 5.10 shows an example of the limiting mechanism.

```
@Query
public EntityCollection<Car> get10Cars() throws PersiJException {
    EntityCollection<Car> cars = new EntityCollection<Car>();
    Car candidateCar = null;
    cars.add(candidateCar);
    Collections.sort(cars, new AscendingComparator(candidateCar.getMake()));
    cars.setMaxSize(10);
    return cars;
}

//Equivalent SQL statement
SELECT * FROM cars ORDER BY make LIMIT 10
```

Figure 5.10: An example of the limiting mechanism in SOQL.

5.6 Modularization

One of the goals we want to achieve with SOQL is the ability to modularize queries. Since the basic encapsulation of a query is a method, modularization can be achieved by letting `@Query` annotated methods be called within each other, and compile time translating this into one query.

This is achieved by letting certain methods on the EntityCollection have special semantics. Instead of calling the methods, the queries expressed in the `@Query` methods that generate them are incorporated in the query currently being compiled.

A prerequisite for modularizing queries like this, is the availability of the source code of the `@Query` methods that are called by the one being compiled currently.

By compiling everything to one SQL statement, query optimization by the RDBMS is possible - the (much worse) alternative being that all the queries are performed separately, all objects instantiated and the methods on the EntityCollections evaluated in the runtime environment.

In the following text we base several examples on a common class illustrated in Figure 5.11.

The following sections will describe each of the methods on EntityCollection that has special semantics. Their functionality is illustrated with simplistic examples. In Section 6.9 a more precise explanation and their translation is given. This section is therefore just a “read and understand examples” section.

5.6.1 Set operations

The familiar set operations intersection, union, and complement can all be modeled by methods defined in the `Collection` interface. The normal behavior of these methods, had they been executed in Java, can be translated to equivalent SQL.

```
class A {
    @Query
    static EntityCollection<Car> getBlackCars() throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        if(candidateCar.getColor().equals("Black")) {
            cars.add(candidateCar);
        }
        return cars;
    }

    @Query
    static EntityCollection<Car> getRedCars() throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        if(candidateCar.getColor().equals("Red")) {
            cars.add(candidateCar);
        }
        return cars;
    }
}
```

Figure 5.11: Common classes for modularization examples.

5.6.1.1 Intersection - retainAll

The API documentation of Java 1.5 has the following to tell about the method `retainAll`:

> *Retains only the elements in this collection that are contained in the specified collection (optional operation). In other words, removes from this collection all of its elements that are not contained in the specified collection.*

If two collections are considered as multi-sets, calling `retainAll` on one with the other as parameter can be considered the intersection of the two multi-sets. SQL has an `INTERSECT` keyword to select the intersection of two select statements. Figure 5.12 shows an example of how this translation can be made.

5.6.1.2 Union - addAll

The Java API documentation on the `addAll` method:

> *Adds all of the elements in the specified collection to this collection (optional operation). The behavior of this operation is undefined if the specified collection is modified while the operation is in progress.*

```
class RetainAll {
    @Query
    public EntityCollection<Car> getSomeCars() throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        if(candidateCar.getPrice().equals("High") {
            cars.add(candidateCar);
        }
        cars.retainAll(A.getBlackCars());
        return cars;
    }
}

// Equivalent SQL statement - intersection
SELECT * FROM cars WHERE price = 'High'
INTERSECT
SELECT * FROM cars WHERE color = 'Black'
```

Figure 5.12: Example using the `retainAll` method.

> *(This implies that the behavior of this call is undefined if the specified collection is this collection, and this collection is nonempty.)*

Again, considering two collections multi-sets, the equivalent set operation of calling `addAll` on one collection with the other as parameter, is an union. Just like intersection, SQL has an `UNION` keyword that can produce the union of two selects. Figure 5.13 shows an example.

5.6.1.3 Difference - removeAll

The Java API documentation on the `removeAll` method is:

> *Removes all this collection's elements that are also contained in the specified collection (optional operation). After this call returns, this collection will contain no elements in common with the specified collection.*

Considering two collections as multi-sets, calling `removeAll` on one with the other, is equivalent to obtaining the complement of the argument collection - or the set difference. The SQL keyword `EXCEPT` covers the same operation, and Figure 5.14 illustrates an example of this.

5.6.2 Subqueries - contains

The `contains` method on a collection object is meant to determine whether a given object is contained within the collection - from the Java API documentation:

```
class AddAll {
    @Query
    public EntityCollection<Car> getSomeCars() throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        if(candidateCar.getPrice().equals("High") {
            cars.add(candidateCar);
        }
        cars.addAll(A.getBlackCars());
        cars.addAll(A.getRedCars());
        return cars;
    }
}

// Equivalent SQL statement - union
SELECT * FROM cars WHERE price = 'High' UNION
SELECT * FROM cars WHERE color = 'Black' UNION
SELECT * FROM cars WHERE color = 'Red';
```

Figure 5.13: Example using the `addAll` method.

```
class RemoveAll {
    @Query
    EntityCollection<Car> getSomeCars(String make) throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        if(candidateCar.getPrice().equals("High")) {
            cars.add(candidateCar);
        }
        cars.removeAll(A.getBlackCars());
        return cars;
    }
}

// Equivalent SQL statement - complement (set difference)
SELECT * FROM cars WHERE price = 'High'
EXCEPT
SELECT * FROM cars WHERE color = 'Black';
```

Figure 5.14: Example using the `removeAll` method.

Returns true if this collection contains the specified element. More formally, returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e)).

In SQL this can be modeled either by using `IN` and a sub-select, and Figure 5.15 shows an example.

```
class B {
    @Query
    static EntityCollection<Model> getOldModels() throws PersiJException {
        EntityCollection<Model> models = new EntityCollection<Model>();
        Model candidateModel = null;
        if(candidateModel.getStartYear() < 1980) {
            models.add(candidateModel());
        }
        return models;
    }
}

// Equivalent SQL statement for getOldModels
SELECT * FROM models WHERE startyear < 1980

class Contains {
    @Query
    static EntityCollection<Car> getSomeCars() throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        if(car.getPrice().equals("High")) {
            if(B.getOldModels().contains(candidateCar.getModel())) {
                cars.add(candidateCar);
            }
        }
        return cars;
    }
}

// Equivalent SQL statement for getSomeCars
SELECT * FROM cars WHERE price = 'High' AND
model IN (SELECT id FROM (
    SELECT * FROM models WHERE startyear < 1980));
```

Figure 5.15: Example using the `contains` method.

5.7 Compilation process

The compilation process of source code using the PersiJ framework is undertaken by JavaC, APT and a compiler for the query methods. Figure 5.16 illustrates the compilation process, and the following describes the process in text.

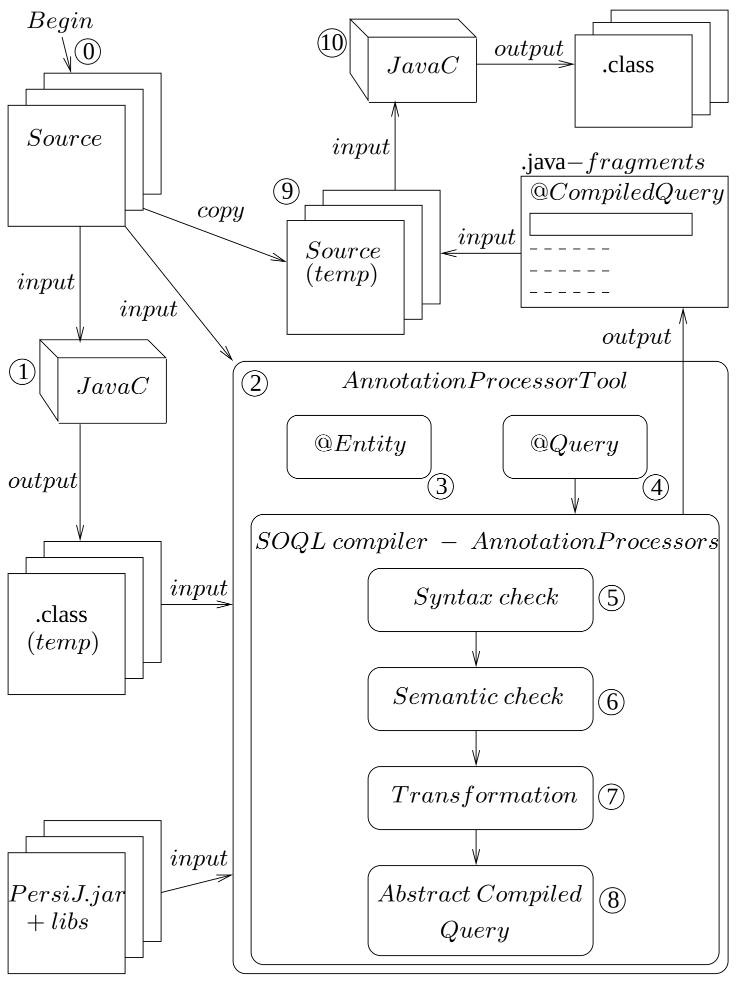


Figure 5.16: The compilation process.

1. **Compile entire source with standard Java compiler:** The first part of the compilation is to compile the source files. The outcome of this process is `.class` files with the whole application.

2. **Use APT to inspect code base:** The next steps are performed using APT with an AnnotationProcessor for each annotation type. When APT finds an annotation, it will automatic call the AnnotationProcessor for that annotation.

   3. **Inspect the classes marked `@Entity`:** This is the first AnnotationProcessor that will run. If no classes are marked with `@Entity`, then we cannot have any query methods, as no database schema is present. Inspecting all classes with this annotation is done to make the schema generation and to build an abstract representation of all the persistable

types.

4. **Inspect the methods marked `@Query`:** Once the abstract representation of all persistable types are in place, it is time to inspect all methods marked with the `@Query` annotation. These will then be passed to the query compiler that does the actual transformation to SQL statements.

    5. **Syntax check:** Using a parser generated with JavaCC from the base grammar, the syntactic check is performed.

    6. **Semantic check:** Some basic semantics are also checked with the parser.

    7. **Transformation:** If the syntactic and basic semantic check completes without errors, the parser will generate an abstract syntax tree, and default visitors (using the visitor pattern [23]). The abstract syntax tree is then used to generate the Abstract Compiled Query (ACQ).

    8. **ACQ:** The ACQ contains the information necessary to generate the SQL queries, and the last compilation step is to use each ACQ to generate a Java fragment for each query method.

9. **Temporary storage of compiled query:** APT does not provide the possibility to replace code in an existing source file. Therefore the source code containing methods marked with the `@Query` annotations are copied to temporary files.

10. **Injecting new source code:** The newly compiled methods (containing the SQL queries) are then injected into the temporarily copies of source files. These are then compiled using JavaC and the compiled code (`.class` files) are then replaced with the old ones (that contained the original compiled code from step 1). This way the original source files are left intact.

6 Transformation of SOQL

In this chapter we will be turning our attention to the informal semantics involving the translation of source code to JDBC and SQL statements. We will also be looking at the template that the newly generated code (the compiled queries marked with `@CompiledQuery`) will be placed in, after the PersiJ compiler has been at work. The translation is described through a number of partial informally defined transformation functions. Functions are given for field dereferencing, method calls, if-statements, sorting, limiting, and modularization.

6.1 Reference sets

Before we go into specific details, we must first make one distinctions – objects are not simply objects in the query method. The object references and primitive values that are in scope for the query method can be divided into two distinct sets. The idea of distinguishing between the two sets is to have one of the sets containing references that runtime are values stored in the database, and the other set then contains references to objects that runtime are on the heap (in memory).

The purpose is then to be able to determine which values can be shipped to the database as parts of query predicates, and which values are to be accessed in the database. Lets just make a definition for these two sets.

Definition 5 (Persistent set) *Contains references to values that runtime are stored in the database.*

Definition 6 (Client set) *Contains references to objects that runtime are on the heap.*

The persistent set can be constructed by following the following steps:

1. Include the reference to the candidate object in the set.

2. When a new reference is returned from dereferencing a field or calling a method, then if the reference is not in either the persistent set or the client set, it can be added to the persistent set subject to these conditions:

   - The object that the new reference points to is a persistable object.
   - The reference to the object on which the member is accessed is already in the persistent set.
   - The member being accessed is not marked with neither `@Transient` nor `transient`.
   - If it is a method being called, it is subject to more conditions which are explained in more detail in Section 6.5.

All the references that are not in the persistent set, are in the client set.

The distinction can also be seen as separating values into those that somehow are dependent of the current candidate object being evaluated in the query predicate, and those values that are not. If a value is independent of the current candidate object, the value can be shipped to the database as a parameter to the query and the expression that yields it, can be executed independently of the query.

These two sets are not actually collected in data structures at runtime, but merely defined to facilitate understanding which statements are legal within SOQL and which are not.

One more definition is needed:

Definition 7 (PersistentExpression) *A PersistentExpression is an expression that yields a reference which is in the persistent set.*

6.2 Template

In this section we will be looking at the template for the translated code. That is, how should we wrap the methods marked `@Query`? Lets start with an example of a `@Query` method. In Figure 6.1 we have an example of a method that retrieves all car objects from the database.

When the PersiJ compiler translates the `@Query` method it will markup the method with `@CompiledQuery`. We want to markup the new method if we need to inspect or use the newly created code.

As already mentioned, we are translating the query methods into JDBC and SQL statements. We therefore need to create a connection to the database as we would do with any application utilizing JDBC. This means that most of the elements present in Figure 6.2 are like the ones we have in Figure 2.1 from Section 2.1.1.

In Figure 6.2 `compiledSqlStatement` in line no. 4 is the part that varies from method to method as this is the SQL statement compiled from the query method. How to build these statements are discussed later in Section 6.6.

```
@Query
public EntityCollection<Car> getAllCars() throws PersiJException {
    EntityCollection<Car> cars = new EntityCollection<Car>();
    Car candidateCar = null;
    cars.add(candidateCar);
    return cars;
}

// Equivalent SQL statement
SELECT * FROM cars;
```

Figure 6.1: Example of a method marked with the `@Query` annotation.

```
@CompiledQuery
public EntityCollection<Car> getAllCars() throws PersiJException {
    // The translated SQL statement
    String compiledSqlStatement = "SELECT * FROM cars";

    Connection conn;
    try {
        conn = EntityCollection.getConnection();
        PreparedStatement pstmt = conn.prepareStatement(compiledSqlStatement);
        ResultSet rs = pstmt.executeQuery();
        EntityCollection cars = new EntityCollection<Car>(rs);
    } catch(Exception e) {
        conn.abort();
        throw new PersiJException(
            "PersiJ encountered an error during querying.", e);
    } finally {
        conn.close();
    }
    return cars;
}
```

Figure 6.2: The translated code from Figure 6.1.

In the original source code (see Figure 6.1) we had the initialization of the car object with `Car candidateCar = null`. As shown in Figure 6.2 the car object itself is nowhere to be found. The reason for this is that it is only used when building up the SQL statements.

6.3 Transformation functions

The rest of this chapter will list a number of partial transformation function definitions for the five functions Φ_{WHERE} (shorthand Φ_W), Φ_{FROM} (shorthand Φ_F), Φ_{ORDER} (shorthand Φ_O), Φ_{LIMIT} (shorthand Φ_L), and Φ_{SET} (shorthand Φ_S) respectively. All functions have legal SOQL code as domain, and legal SQL as range. The function definitions are quite informal, and are used to clarify how the transformation to SOQL takes place.

The final SQL statement used in the compiled query method, is constructed by starting with "`SELECT * FROM` $PrimaryTable$" where $PrimaryTable$ is the name of the table that the class being queried maps to. Then the cumulative output of the Φ_{FROM} function is appended. Thereafter "`WHERE TRUE`" is appended, and the cumulative output from the Φ_{WHERE} function is appended. The output from the rest of the functions are then appended.

If there is an if-statement that introduces branching at runtime (see Section 5.3 and Section 6.6), the output of the transformation of the different branches are stored in separate variables, and runtime the query is combined.

6.3.1 Legend to reading Φ definitions

Text in a font like this: `code` and encapsulated in `"` are literals found directly in the source code, and text like this: $statement$ comes from definitions like Definition 7. In the output of a function, the sign + is used to indicate string concatenation. The input to the Φ functions should be seen as a form of pattern matching.

6.4 Field dereferencing

This section explains what happens when during parsing of the query method, when a *PersistentExpression* is encountered, which is subsequently followed by an access to a field on that persistable object. We use $FieldName$ to denote the name of the field.

Using the abstract representation of the database schema, the name of that field in the database is looked up.

6.4.1 Primitives and Strings

If the field is a primitive, a `String` object or one of the objects wrapping the primitives, the fields values in the database is in the same relation as the persistable object.

Definition 8 (DBFieldName) *The name of the field in the database is denoted as DBFieldName.*

Name	Shorthand	Explanation
$Table$	T	The name of the table that is being dereferenced.
$TablePrimaryKey$	TPK	The name of the column containing the primary key of $Table$.
$ForeignTable$	FT	The name of the table that contains the type of the field in question.
$ForeignKey$	FK	The name of the column that contains the primary key of the other table.

Table 6.1: Legend for Equation 6.2 and Equation 6.3

Equation 6.1 shows what is appended to the `WHERE` part of the SQL statement.

$$\Phi_W\left(\ PersistentExpression\texttt{"."}FieldName\ \right) \rightsquigarrow\ DBFieldName \tag{6.1}$$

6.4.2 One-to-one, one-to-many, owning side

If the field is an object, then there is a relation between the two objects in the database. Recalling from Section 4.3.4 relations between objects have an owning side. If this persistable object is the owning side of a one-to-many or a one-to-one relationship, then Equation 6.2 and Equation 6.3 applies. The table and row names used in the equations are explained in Table 6.1.

$$\begin{gathered}\Phi_F\left(\ PersistentExpression\texttt{"."}FieldName\ \right)\\ \rightsquigarrow\\ \texttt{" JOIN "}+FT+\texttt{" ON "}+\\ T+\texttt{"."}+TPK+\texttt{"="}+FT+\texttt{"."}+FK\end{gathered} \tag{6.2}$$

$$\begin{gathered}\Phi_W\left(\ PersistentExpression\texttt{"."}FieldName\ \right)\\ \rightsquigarrow\\ FT\end{gathered} \tag{6.3}$$

6.4.3 Bidirectional, many-to-one, one-to-one, not owning side

Equation 6.4 and Equation 6.2 (Table 6.2 extends the legend) covers the case when:

- The relation between the objects is a one-to-one and bidirectional, where the current object is not the owning side.
- The relation between the objects is a many-to-one. In this case, the field will be an EntityCollection, and the type mapped to the database will be the type parameter of the EntityCollection.

Name	Shorthand	Explanation
$ForeignTablePrimaryKey$	$FTPK$	The name of the primary key of the foreign table.

Table 6.2: Legend for Equation 6.4

Name	Shorthand	Explanation
$AssociativeTable$	AT	The name of the associative table that maps the relation.
$AssociativeTPK$	$ATPK$	The name of the column that contains the primary key from the table (TPK) we are selecting from.
$AssociativeFTPK$	$AFTPK$	The name of the column that contains the primary key from the foreign table ($FTPK$).

Table 6.3: Legend for Equation 6.4

$$\begin{gathered}\Phi_F\left(\ PersistentExpression\texttt{"."}FieldName\ \right)\\ \rightsquigarrow\\ \texttt{" JOIN "}+FT+\texttt{" ON "}+\\ T+\texttt{"."}+FK+\texttt{"="}+FT+\texttt{"."}+FTPK\end{gathered}\tag{6.4}$$

6.4.4 Many-to-many

In the case that the relation is a many-to-many, the object will be an EntityCollection. In this case, the relation is in the database modeled with an associative table. Equation 6.5 covers this case, and Table 6.3 extends the legend for the equation. Equation 6.3 still applies in this case.

$$\begin{gathered}\Phi_F\left(\ PersistentExpression\texttt{"."}FieldName\ \right)\\ \rightsquigarrow\\ \texttt{" JOIN "}+AT+\texttt{" ON "}+\\ T+\texttt{"."}+PK+\texttt{"="}+AT+\texttt{"."}+ATPK\\ \texttt{" JOIN "}+FT+\texttt{" ON "}+\\ AT+\texttt{"."}+AFTPK+\texttt{"="}+FT+\texttt{"."}+FTPK\end{gathered}\tag{6.5}$$

6.5 Method calls

If a method call is encountered during the parsing of the body of a query method, the following conditions and associated actions apply.

- If the callee of the method is either the candidate object or a persistable object present in the persistent set. A method call on this reference can be divided into two distinct groups:

    1. The method is a prototypical `get` method, which does nothing else than return either a primitive value or a reference to a persistable object, and has no arguments. If this is the case, this method call corresponds to a join in the translated code. The reference returned from the method call is added to the persistent set. The transformation rules are similar to that of field dereferencing, and are covered by Equation 6.2 through Equation 6.5.
    2. The method is not a prototypical `get` method, and while the return type is either persistable or a primitive type, the method does more than just return - and the method may have arguments. If this is the case, a byte-code analysis of the method is required to determine whether the logic it contains can be rewritten as a part of the SQL statement. In this case, it also has to be determined whether the method is free of side-effects, and employing a technique like mentioned in Section 5.1.5.7 may prove useful. In any case, determining to which degree it is possible to make a transformation for these method calls is an open question.

- If the callee of the object is not either the candidate object or a persistable object present in the persistent set:

    1. The method does not take an argument that is in the persistent set, and no previous method has been invoked on the object with an argument that is in the persistent set. If this is the case, invocation of the method may be moved unaltered to the translated query method.
    2. The method takes an argument that is in the persistent set. Byte-code analysis of the method body or some side-effect analysis must be employed to determine whether the outcome of the method is dependent on the argument. If it is independent, the invocation may be moved to the translated block, with the argument replaced by null or similar. In the case that is is dependent on the argument, it is an open question whether a translation can be performed.

6.6 If-statements

We will now be looking at the if-statements. These are the ones that build up the query inside the body of the method. If the expression of the if-statement contains a reference from the persistent set, then there are a few transformation rules that apply.

Definition 9 (ifPredicate) *An ifPredicate is the expression in an if-statement that evaluates to true or false.*

The *ifPredicate* is, in other words, the predicate that the if-statement evaluates. Normal Java syntax applies to the predicate.

Definition 10 (addStatement) *An add statement is on the form <Identifier>.add(<Identifier>); where the first identifier is the EntityCollection, and the second identifier is the entity.*

When a branch of the if-statement does not contain an *addStatement*, there is no reason to use the rest of that branch of the if-statement to build up the query as whichever predicates might be expressed are not to be used in the final predicate. Take for instance the pseudo code in Figure 6.3 - the tree below *predicate 3* does not need to be evaluated as no matter if *predicate 4* evaluates to true or false, it will always be added due to the `add()` in line no. 9. The entire else-statement (line no. 11) does not need to be evaluated as there is not found any add-statement in this part of the tree.

```
 1 // Pseudo code of if-statements
 2 if(predicate 1) {
 3     if(predicate 2) {
 4         if(predicate 3) {
 5             if(predicate 4) {
 6                 add();
 7             }
 8         }
 9         add();
10     }
11 } else {
12     if(predicate 5) {
13         // Do nothing containing an add method
14     }
15 }
```

Figure 6.3: Example of tree structure of if-statements.

Definition 11 (thenStatement) *A thenStatement is the first part of the if-statement, which in normal Java will be executed if the ifPredicate evaluates to true. The contents of the thenStatement is the productions of the grammar non-terminal "Statement".*

Definition 12 (elseStatement) *An elseStatement is the second part of the if-statement, which in normal Java will be executed if the ifPredicate evaluates to false. The contents of the elseStatement is the productions of the grammar non-terminal "Statement".*

These two definitions are quite important to understand as they will form the foundation for the query. Definition 11 applies to the *ifPredicate* evaluating to true (it stems from the term *if-then*). The *elseStatement* is used when the *ifPredicate* evaluates to false.

To make the following easier to understand we have added the definitions of the existence and non-existence of the *addStatement*.

Definition 13 (Existence of addStatement) *The existence of an addStatement within an thenStatement or an elseStatement is written as* $\exists add$*.*

Definition 14 (Non-existence of addStatement) *The non-existence of an addStatement within an thenStatement or an elseStatement is written as* $\neg\exists add$*.*

Having these definitions in place, we proceed describing the translation of if-statements in SOQL. There are four cases, dependent of the existence of an $addStatement$ in the $thenStatement$ and/or the $elseStatement$. The four cases are shown in equations 6.6 through 6.9.

$$\Phi_W\left(\begin{array}{l}\texttt{"if("}ifPredicate\texttt{") \{"}\\ \quad thenStatement \mid \neg\exists add\\ \texttt{"\} else \{"}\\ \quad elseStatement \mid \neg\exists add\\ \texttt{"\}"}\end{array}\right) \rightsquigarrow \texttt{""} \tag{6.6}$$

$$\Phi_W\left(\begin{array}{l}\texttt{"if("}ifPredicate\texttt{") \{"}\\ \quad thenStatement \mid \exists add\\ \texttt{"\} else \{"}\\ \quad elseStatement \mid \neg\exists add\\ \texttt{"\}"}\end{array}\right) \rightsquigarrow \begin{array}{l}\texttt{"("+}\Phi(ifPredicate)\texttt{+}\\ \texttt{"AND"+}\Phi(thenStatement)\texttt{+")"+}\end{array} \tag{6.7}$$

$$\Phi_W\left(\begin{array}{l}\texttt{"if("}ifPredicate\texttt{") \{"}\\ \quad thenStatement \mid \neg\exists add\\ \texttt{"\} else \{"}\\ \quad elseStatement \mid \exists add\\ \texttt{"\}"}\end{array}\right) \rightsquigarrow \begin{array}{l}\texttt{"(NOT"+}\Phi(ifPredicate)\texttt{+}\\ \texttt{"AND"+}\Phi(elseStatement)\texttt{+")"}\end{array} \tag{6.8}$$

$$\Phi_W\left(\begin{array}{l}\texttt{"if("}ifPredicate\texttt{") \{"}\\ \quad thenStatement \mid \exists add\\ \texttt{"\} else \{"}\\ \quad elseStatement \mid \exists add\\ \texttt{"\}"}\end{array}\right) \rightsquigarrow \begin{array}{l}\texttt{"("+}\Phi(ifPredicate)\texttt{+}\\ \texttt{"AND"+}\Phi(thenStatement)\texttt{+")"+}\\ \texttt{"OR"+}\\ \texttt{"(NOT ("+}\Phi(ifPredicate)\texttt{+}\\ \texttt{") AND"+}\Phi(elseStatement)\texttt{+")"}\end{array} \tag{6.9}$$

If several if-statements are present on the outermost level, each of their transformations is joined with an `OR`.

If there are if-statements where the $ifPredicate$ does not contain any references from the persistent set, the predicates that might be expressed deeper within this if-statement, are combined dependent on the runtime evaluation of the $ifPredicate$, and the resulting control flow. Figure 6.4 shows an example, where the query actually runtime executed against the database, is dependent on the runtime execution branch.

```
public EntityCollection<Car> getSomeCars(boolean switch)
                                              throws PersiJException {
    EntityCollection<Car> cars = new EntityCollection<Car>();
    Car candidateCar = null;
    if(switch) {
        if(candidateCar.getColor().equals("Yellow")) {
            cars.add(candidateCar);
        }
    } else {
        if(candidateCar.getPrice().equals("High")) {
            cars.add(candidateCar);
        }
    }
    return cars;
}

/**
This part of the SQL string is not dependent on the runtime value of switch:
    SELECT * FROM cars WHERE
Dependent on the runtime value of switch, the independent string is then
concatenated with either:
    cars.color = 'Yellow'
or
    cars.price = 'High'
**/
```

Figure 6.4: Example of control flow to build query predicates dependent at runtime execution.

6.7 Sorting

As discussed in Section 5.4, sorting can be implemented by assigning special semantics to a call on `java.util.Collections` `sort` method. This call must be on the outermost level of the query method.

Definition 15 (ResultIdentifier) *The ResultIdentifier is the identifier defined in the first line of the QueryMethod, which points to an EntityCollection that is ultimately returned from the query.*

Definition 16 (FieldIndicator) *A FieldIndicator is either an expression that yields and object assignable to* `java.lang.member`*, or a field dereferencing on the candidate object, or a call to an accessor method on the candidate object.*

Definition 17 (PersiJComparator) *A PersiJComparator is a class name that is either* `AscendingComparator` *or* `DescendingComparator`.

Definition 18 (SortOrder) *A SortOrder is either* `ASC` *or* `DESC`.

Equation 6.10 shows the transformation function for sorting.

$$\Phi_O\left(\begin{array}{l}\texttt{"Collections.sort("}ResultIdentifier\texttt{","}\\ \texttt{"new"}PersiJComparator\texttt{"("}FieldIndicator\texttt{"))"}\end{array}\right) \\ \leadsto \\ \texttt{" ORDER BY "}{+}DBFieldName{+}SortOrder \tag{6.10}$$

Prerequisites for this transformation function is that the field to be sorted on has an ordering, and that it is either a primitive or a string.
The chosen $SortOrder$ depends on which $Comparator$ is chosen.

6.8 Limiting

To limit the maximum number of objects returned from the query, the method `setMaxSize` can be called on the $ResultIdentifier$. The call must be present at the outermost level of the query method. Equation 6.11 shows the transformation function.

Definition 19 (IntegerExpression) *An IntegerExpression does not contain any references from the persistent set, and yields either the primitive* `int` *or an object of type* `java.lang.Integer`.

$$\Phi_L\left(ResultIdentifier\texttt{".setMazSize("}IntegerExpression\texttt{");"} \right) \\ \leadsto \\ \texttt{" LIMIT "}{+}Eval(IntegerExpression) \tag{6.11}$$

$Eval(IntegerExression)$ denotes that the *IntegerExpression* is evaluated runtime, and the result inserted in the query.

6.9 Modularization

As we reviewed in Section 5.6, the methods `contains`, `removeAll`, `addAll`, and `retainAll` can be used to modularize queries. In this section we proceed by establishing a few definitions, and prerequisites for an informal definition of a transformation function for these methods.

Definition 20 (QueryExpression) *An QueryExpression is an expression that yields an EntityCollection as a result of a call on a method that is marked with the* `@Query` *annotation.*

Definition 21 (ExternalQueryMethod) *The ExternalQueryMethod is the source code of the implementation of the method that is called in the QueryExpression.*

Definition 22 (CandidateObjectIdentifier) *The CandidateObjectIdentifier is the identifier pointing at the candidate object.*

6.9.1 Set operations

The transformation function definitions in Equations 6.12 through 6.14 define the transformation functions for respectively intersection, union, and complement. They are all subject to the following prerequisites:

- The statement they appear in must be at the outermost block level of the method. This means that they cannot be nested within an if-statement.
- The source code for the query method of the $QueryExpression$ must be present at compile-time.
- The argument to the method being called on the $ResultIdentifier$ (these are `retainAll`, `addAll`, and `removeAll`) must be parametrized with the same type as the $ResultIdentifier$.

The result of the transformation function is to be appended to the SQL string being built for the current query. The expression $\Phi(ExternalQueryMethod)$ is the SQL query string that can be compiled for the $ExternalQueryMethod$. Another part of the transformation that is not covered in the equations is the surrounding statements, which have to be merged too, possibly renaming identifiers, etc. to avoid naming conflicts.

$$\Phi_S\left(\begin{array}{l} ResultIdentifier\texttt{".retainAll("} \\ \quad QueryExpression\texttt{");"} \end{array}\right) \leadsto \begin{array}{l} \texttt{" INTERSECT "+} \\ \Phi(ExternalQueryMethod) \end{array} \tag{6.12}$$

$$\Phi_S\left(\begin{array}{c} ResultIdentifier\texttt{".addAll("} \\ QueryExpression\texttt{");"} \end{array}\right) \rightsquigarrow \begin{array}{l} \texttt{" UNION "+} \\ \Phi(ExternalQueryMethod) \end{array} \tag{6.13}$$

$$\Phi_S\left(\begin{array}{c} ResultIdentifier\texttt{".removeAll("} \\ QueryExpression\texttt{");”} \end{array}\right) \rightsquigarrow \begin{array}{l} \texttt{" EXCEPT "+} \\ \Phi(ExternalQueryMethod) \end{array} \tag{6.14}$$

6.9.2 Subqueries - contains

As we showed in example Figure 5.15, the `contains` method on the EntityCollection can be used to modularize queries and test for existence of a value in another query.

The transformation of a `contains` call is subject to some prerequisites. The call must be made to an EntityCollection that is not the one being returned from the current query. Moreover, the object on which it is called, must be returned from a $QueryExpression$, and the source code for the method must be present at call time. The call must be placed within an $ifPredicate$. The transformation is two-fold. The `FROM` part of the query needs to be amended with the necessary join (see Equation 6.2 through Equation 6.5). Equation 6.15 takes care of adding a predicate to the `WHERE` part of the query.

$$\begin{array}{l} \Phi_W\left(QueryExpression\texttt{".contains("}PersistentExpression\texttt{");"} \right) \\ \qquad\qquad\qquad\qquad \rightsquigarrow \\ ForeignKey\texttt{+" IN(SELECT "+}PrimaryKey\texttt{+} \\ \texttt{" FROM ("+}\Phi\left(ExternalQueryMethod\right)\texttt{+"))"} \end{array} \tag{6.15}$$

6.10 Implementation

As stated in the method description (see Section 3.5), a compiler prototype has been built alongside with the language design process. The current state of the compiler includes all steps until the semantic check (step 7 in Figure 5.16).

Parts of the semantic check and transformation process has been implemented, but the compiler is not yet in a working state.

7 Discussion

In this chapter we will go through the criteria presented in the analysis that can be used to evaluate a method of querying, and evaluate SOQL with respect to these. We will also be looking at the expressiveness of SOQL vs. SQL.

7.1 Evaluating by criteria

Throughout the evaluation we will not only refer to SOQL, but also to PersiJ because that fulfilling some criteria or not is not only due to properties solely in SOQL, but often also because of the combined properties of PersiJ and SOQL.

7.1.1 Static checking

Static checking consists of two facets:

Static type checking The ability to make static type checking of the interactions between the programming language (in this case Java) and the RDBMS. In other words, it is at compile time possible to check whether there exists any type-related mismatches between the queries passed from the application to the database.

Static semantic checking The ability to assure at compile-time that only existing tables and rows are referenced in the program source, and assure compile-time that the SQL statements that are executed runtime are sound statements.

PersiJ (and therefore also SOQL) operates on a database schema that is determined based upon the metadata provided by the developer (explicitly or implicitly) in the model classes in the form of annotations. Since all queries to the database are generated based upon interactions between objects in Java, the referenced tables and rows are sure to exist in the database - subject to the prerequisite that the mapping metadata is correct.

Queries in SOQL are not string based, and this means that parameters to queries are by the programmer expressed using existing classes and objects in a strongly, static typed manner.

Moreover, this also means that the static type checking done by the Java compiler also applies to the SQL statements. If the source code is accepted by the Java compiler, the generated SQL is also free of type errors (i.e. there is no mismatch between types in the generated SQL statements).

One problem still remains - it is still possible to do script injection by placing malicious strings as parameter content. This problem can be alleviated by using existing approaches [9] combined with the PersiJ framework.

In summary, both static semantic and type checking is done in SOQL due to inherent properties of the query language design.

7.1.2 Automatic marshalling and unmarshalling

The PersiJ framework provides automatic marshalling and unmarshalling of objects when working with a persistent storage. This is handled by the EntityCollection during object instantiation, and when storing data in the database. This design decision imposes a restriction - it is not possible to query data that is not mapped to objects.

7.1.3 Same paradigm as host language

SOQL does not contribute with new language constructs, and it works with (a subset of) the syntax of Java. The semantics are quite another story, as they deviate somewhat from Java.

Consider Figure 7.1, which is a simple example of a query searching for all cars by the model Ford. This code is syntactically compliant with Java, and the Java compiler will find the code to be semantically sane. But reading the code and trying to understand what happens applying understanding of normal Java semantics only, will prove a futile exercise. Take line no. 4 which is semantically correct, but in line no. 5 we are using the `candidateCar` object to check, if its model equals Ford - and considering that the object is `null`, this should throw a `NullPointerException`. This is not intuitive and requires knowledge to the SOQL language in order to understand that the candidate object (Definition 3) is used as base for the SQL statement and is in principle iterated over once for each candidate row in the database. The query is sane in SOQL - and will produce valid code once compiled with the PersiJ compiler.

Figure 7.1 addresses another problem with the semantics. Normal Java semantics for if-statements is branching of code execution subject to the evaluation of the predicate. In SOQL there is no execution flow per se, and therefore no branching as such - instead, the if-statements are used to build predicates, and dependent on the existence of an `add()` statements the different branches have different impact on the query predicate being built. Moreover, if no `add()` statement is present within a branch, branch may be entirely ignored (e.g. see Section 6.6). Branching is not altogether non-existent, since if-statements which test an expression that uses no values from the database, can be used to runtime determine construction of the query

```
 1 @Query
 2 public EntityCollection<Car> getFord(Coutner cntr) throws PersiJException {
 3     EntityCollection<Car> cars = new EntityCollection<Car>();
 4     Car candidateCar = null;
 5     if(candidateCar.getModel().equals("Ford")) {
 6         cars.add(candidateCar);
 7     } else {
 8         // cntr.count();
 9     }
10     return cars;
11 }
12
13 // Equivalent SQL statement for getFord
14 SELECT * FROM cars WHERE model = "Ford"
```

Figure 7.1: Example searching for all types of Ford.

- but the difference of the semantics dependent on the contents of the expression being tested is not obvious from the syntactic construction (`if(Expression) Block [else Block]`) since it is the same.

The syntactical constructions that can actually be translated by the SOQL compiler are subject to a number of prerequisites. For the developer these prerequisites will probably seem intricate and hard-to-understand, and contribute to SOQL being perceived as non-intuitive. To illustrate this, take the statement in line no. 9 in Figure 7.1. If it was not commented-out, it would not be possible to translate it to an SQL query. Why? There are many valid semantic questions to ask and no simple answer: How many times should the statement be executed? Once for each tested tuple in the database (and how many is that)? Should it be executed once for each tuple that does not match the predicate? If it should be executed at all, how is it possible to determine the number of executions without instantiating all candidate objects?

The clarity of what can and what cannot be translated into a SQL query becomes even worse when methods are called on objects from the persistent set, where the methods are not simply accessor methods.

In summary, although SOQL is syntactically within Java, the semantics are significantly different, and especially the lack of transparent control flow and intricate rules determining legal and illegal expressions takes SOQL quite far from the semantics of the host language.

7.1.4 Minimal verbosity

One of the first design decisions for SOQL was not to follow the Native Queries approach, where an implementation of a class was necessary, as we felt this required too much syntactical overhead. In retrospect, the final form of query methods in

SOQL is not significantly less verbose.

Comparing the verbosity of SOQL to writing the same query in JDBC, SOQL is less verbose. Illustrating this for a quite simple query is Figure 7.2. The SQL part of the JDBC code is of course much less verbose than the SOQL query method, but comparing SOQL to SQL is not quite fair, since the compiled SOQL statement does much more for the programmer than just being a SQL statement.

```
class A {
    @Query
    public static EntityCollection<Owner> getYoungCarOwners()
                                                throws PersiJException {
        EntityCollection<Owner> owners = new EntityCollection<Car>();
        Owner candidateOwner = null;
        if(candidateOwner.getBirthYear() > 1976) {
            owners.add(candidateOwner);
        }
        return owners;
    }

    // Equivalent JDBC method for getYoungCarOwners
    public ResultSet getYoungCarOwnersInJdbc() {
        ResultSet rs = null;
        Connection conn;
        try {
            conn = EntityCollection.getConnection();
            PreparedStatement pstmt = conn.prepareStatement(
                "SELECT * FROM owners WHERE birthyear > ?");
            pstmt.setInt(1, 1976);
            rs = pstmt.executeQuery();
        } catch(Exception e) {
            conn.abort();
            // Do exception handling
        } finally {
            conn.close();
        }
        return rs;
    }
}
```

Figure 7.2: Example of a simple query.

SOQL can also use other existing queries within a query, and compile them all into one coherent SQL query. Consider the queries of Figure 7.2 and Figure 7.3 that are used in Figure 7.4

Lets say that we want to combine the three queries from classes `A` and `B` in such a fashion that we get all the cars whose owners are born after 1976 and the models of the cars that have been involved in most car crashes in union with the most expensive cars. The method for this is called `getHighRiskCars` and is illustrated in Figure 7.4.

```
class B {
    public static EntityCollection<Model> getTopCrashedModels(int numResults)
                                                        throws PersiJException {
        EntityCollection<Model> models = new EntityCollection<Model>();
        Model candidateModel = null;
        models.setMaxSize(numResults);
        models.add(candidateModel);
        Collections.sort(models, new DescendingComparator(
                candidateModel.getNumCrashed()));
        return models;
    }

    // Equivalent SQL statement for getTopCrashedModels
    SELECT * FROM models ORDER BY numcrashed DESC LIMIT 10

    public static EntityCollection<Car> getMostExpensiveCars(int numResults)
                                                        throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        cars.add(candidateCar);
        cars.setMaxSize(numResults);
        Collections.sort(cars, new DescendingComparator(cars.getPrice()));
        return cars;
    }

    // Equivalent SQL statement for getMostExpensiveCars
    SELECT * FROM cars ORDER BY price DESC LIMIT 10
}
```

Figure 7.3: Example of simple queries.

Writing queries in SQL can be troublesome, and debugging a statement like the one in Figure 7.4 can be even more troublesome. Since SOQL provides the ability to modularize queries, these can be reused in different contexts, and the modularization provides with better error-localization (i.e. if it is known that one query method works correctly, it can safely be assumed that it also works correctly within the new context). As the complexity of the query grows it becomes increasingly harder to read in SQL, whereas in PersiJ it remains readable due to the modularization.

```
class C {
    @Query
    public EntityCollection<Car> getHighRiskCars() throws PersiJException {
        EntityCollection<Car> cars = new EntityCollection<Car>();
        Car candidateCar = null;
        if(A.getYoungCarOwners().contains(candidateCar.getOwner())) {
            cars.add(candidateCar);
        } else {
            if(B.getTopCrashedModels(10).contains(candidateCar)) {
                cars.add(candidateCar);
            }
        }
        cars.Addall(B.getMostExpensiveCars(10));
        return cars;
    }
}

// Equivalent SQL statement for getHighRiskCars
SELECT * FROM cars JOIN owners ON
  (cars.ownerid = owners.id WHERE cars.ownerid IN
    (SELECT id FROM (SELECT * FROM owners WHERE birthyear > 1976))
  OR (cars.id IN (SELECT id FROM
    (SELECT * FROM models ORDER BY numcrashed DESC LIMIT 10))))
UNION
  (SELECT * FROM cars ORDER BY price DESC LIMIT 10)

```

Figure 7.4: Example of modularized query method.

7.1.5 Minimal language alteration

As just mentioned in Section 7.1.3 the PersiJ framework operates within the Java programming language without introducing new keywords, language constructs, or paradigms as SQLJ and C# does.

This has been done at great expense. Although no new language elements have been introduced, imposing the restriction of not allowing any new language elements and at the same time rejecting the Native Queries approach results in a fashion of querying that albeit being syntactically compliant with Java is not understandable without a prior knowledge to the special semantics of SOQL.

Native Queries was not considered as a way of doing querying as the requirement of making an (anonymous) implementation of the `Predicate` class in Java is somewhat unwieldy, but this is simply because the Java programming language does not have a language construct like delegates. Delegates are existent in C#. Native Queries in C# is quite elegant, and easy to understand and read. Recall Figure 3.1 found in Section 3.1.3. Writing this in C# with delegates is illustrated in

Figure 7.5.

```
delegate(Student student)
    return student.Age < 20
        && student.Name.Contains("f");
}
```

Figure 7.5: Re-writing the query part of Figure 3.1 to C# (taken from [14])

It is possible to implement delegates into the Java language using the Adapter pattern [23]. This approach has never really been practiced in Java as it is unwieldy. It has also been argued that inner classes provide the same functionality as delegates [42], and the delegate construct is therefore not needed when programming in Java.

In summary, although SOQL meets the criteria of minimal language alteration syntactically, it semantically deviates very much from Java. Altering SOQL by introducing new language constructs that requires the programmer to learn new semantics might be a better alternative than sticking strictly to a no-new-language-constructs design.

7.1.6 Modularization

As discussed earlier, SOQL does have modularization capabilities. As sketched, they are now restricted to set operations and using an equivalent of the `IN` SQL operator in query predicates. Although simple, these modularization capabilities does provide the system developer with the possibility of splitting queries into smaller chunks that can be combined in different contexts while at runtime retaining the efficiency of one large SQL query.

Figure 7.4 is an example of this. Although it is possible to combine and thereby modularize these queries, modularization statements are subject to a number of prerequisites. These prerequisites are somewhat unclear to the developer without intimate knowledge of the semantics of SOQL.

There is one deficiency that might need clarification. Since the queries are effectively merged at compile-time, replacement of code in the `.class` files subsequently will possibly void the intent of the replacement, since changes in query methods will not propagate to the queries in which they are used. Propagating them will require a full recompile.

As will be discussed in Section 7.2, SOQL still lacks constructs to mirror some modularization operations that are present in SQL.

In summary, the modularization features of SOQL may be extended with more constructs to express e.g. universal and existential quantification, and SOQL would benefit from a greater transparency of legal ways to use modularization constructs.

7.1.7 Optimization

All queries expressed in SOQL are compiled to SQL statements, and they are shipped to the RDBMS. All optimization hereafter is left as an exercise for the database. As mentioned, it is possible to modularize queries. When these modularized query methods are compiled, they will also produce a single query (see for instance Figure 7.3). This way the RDBMS can optimize the SQL statement. Criteria are also shipped separately to the database. It is also possible to implement some prefetching techniques in EntityCollection. In short, the optimization criteria has been met.

Further optimization by letting the system developer have the option of rewriting the compiled queries into (possibly more efficient) equivalent SQL statements is not possible. This could be made possible by letting the system developer inspect and alter the Java fragments with the compiled queries before they are inserted into the source code in the final stages of the compilation process. However, these changes will not propagate back to the original source and simply disappear once the original source code is compiled once more.

The system developer may still regain detailed control. It is possible to use handwritten SQL statements. In order to use these the developer must use the `getConnection()` method to manually open and use the JDBC connection (e.g. see Section 4.4.4).

7.2 Expressiveness of SOQL vs. SQL

An interesting question to pose is the expressiveness of SOQL compared to the query part of SQL. It is quite clear that SOQL is not as expressive as SQL. The following lists some things that can be expressed in SQL but not in SOQL:

Null values Null values have quite different semantics in Java and SQL. In Java a null pointer basically means undefined, while in SQL it means unknown [35]. Null values cannot be operated on in Java. Trying to call a method to a null pointer throws an exception. In SQL nulls have special semantics - e.g. in conjunction with comparison operations. The design of SOQL does not take null values into consideration, and if it is to operate on legacy data, it is a necessity to define a behavior.

Aggregates As laid out in this project, it is not possible to use aggregate functions and do grouping with SOQL. Extending SOQL to make this possible to some degree may not be too hard for grouping functions that are easily mapped to methods in the EntityCollection in the same style as `addAll`, etc.

Select operations In the select clause of a select statement it is possible to do much more than just retrieve all rows for a table - e.g. `SELECT make, model, price * 1.25 AS inclvat FROM cars` where the value of the price row is multiplied by 1.25 before being returned. This is not possible in SOQL where all queries select all rows from the table being queried. Due to this restriction, it is not possible to perform calculations in the database, but they have to be performed by the application instead.

Non-mapped data SOQL always performs queries that return objects that are mapped into the database. Querying across several tables and returning a result-set that does not necessarily map to an entity is not possible. An extension of this restriction is that other joins than the straightforward join (outer join, inner join, full join, etc.) cannot be expressed.

Stored procedures and user defined functions Most RDBMSs have the ability to write stored procedures and user defined functions that may be used in queries, etc. These cannot be used with SOQL.

8 Conclusion

8.1 Conclusion

With this project we wish to answer the question posed in the problem statement (Section 3.6):

> *Is it possible to design an object-oriented query language integrated with Java, which can be transformed to Java code that ships the query as SQL to the database while still fulfilling the criteria: static checkable, automatic marshalling and unmarshalling, minimal verbosity, minimal language alteration, modularizable, and optimizable?*

In summary of the previous chapter, this is how the designed query language Simple Object Query Language (SOQL) fulfills the selected criteria:

Static checking Both static type and semantic checking. Script injection problems can be remedied with existing techniques. This criteria *is fulfilled*.

Automatic marshalling and unmarshalling *Is fulfilled*.

Same paradigm as host language Although SOQL in the syntactical sense is squarely within the object-oriented paradigm, the special semantics used in conjecture with building predicates deviate so much from standard Java semantics that we consider this criteria as *not fulfilled*.

Minimal verbosity Although less verbose than using JDBC and SQL statements, SOQL is not less verbose than Native Queries. This criteria is *partly fulfilled*.

Minimal language alteration Syntactically there is no alteration to the host language, but semantics deviate so much from Java semantics that the language is altered, and this criteria is *not fulfilled*.

Modularization It is in limited ways possible to modularize and combine queries without sacrificing performance, since it is possible to combine the modularized queries to one request to the database. More ways of modularizing can be imagined, and we consider this criteria *is fulfilled*.

Optimization In the sense that the database is free to optimize queries, and that modularized queries are combined to one, this criteria *is fulfilled*. In the sense that it is not possible for the system developer to tweak the generated SQL queries, this criteria is *not fulfilled*.

Most of the criteria set for SOQL have been fulfilled. Further development, refinement, and implementation of a working compiler we believe would provide with a query language that could be useful for a large subset of applications. Especially solving the static checking issues while retaining the ability to ship whole queries to the database and modularize queries, etc., might even make SOQL a favorable alternative to other query languages.

However, since designing a query language that is integrated with the host language is the main goal of this project, not fulfilling the criteria of being in the same paradigm, is a serious drawback. Unfortunately, the differences of semantics of SOQL and Java are intricate. For this reason, we do not believe that using SOQL makes querying easier to express than using existing alternatives.

8.2 A broader perspective

The conclusion leads to a consideration about whether or not it is appropriate to express queries on relational data using object-oriented syntax and semantics. It is clear that deficiencies regarding e.g. lack of static checking of string-based query forms must be solved, but we do not necessarily believe that this solution at the same time should grow closer to the host language regarding syntax and semantics.

If the persistent storage mechanism is an object-oriented database, the case may be different. When the persistent storage matches the language of the application, querying object-oriented databases with an object-oriented language may prove just right.

Which paradigm to use in the different tiers of an n-tiered application may be a matter of choosing the right tool for the right job. The object-oriented paradigm expresses behavior and state combined very well - but may just not be very well suited for expressing queries on data (that does not incorporate behavior). SQL is a partly declarative language, and we speculate that maybe using a declarative-based language is a better tool for expressing queries on relational databases?

Bibliography

[1] ABRAHAM SILBERSCHATZ, H. F. K., AND SUDARSHAN, S. *Database System Concepts*, fourth edition ed. McGraw-Hill, 2002.

[2] AMBLER, S. W. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley, John & Sons, Incorporated, 2003.

[3] ATKINSON, M., AND JORDAN, M. Orthogonal Persistence for the Java Platform - Draft Specification, jun 24 1999.

[4] ATKINSON, M. P. Persistence and Java - A Balancing Act. In *Proceedings of the International Symposium on Objects and Databases* (London, UK, 2001), Springer-Verlag, pp. 1–31.

[5] ATKINSON, M. P., AND JORDAN, M. J. Issues Raised by Three Years of Developing PJama: An Orthogonally Persistent Platform for Java. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory* (London, UK, 1999), Springer-Verlag, pp. 1–30.

[6] BERNSTEIN, P. A., PAL, S., AND SHUTT, D. Context-Based Prefetch for Implementing Objects on Relations. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), Morgan Kaufmann Publishers Inc., pp. 327–338.

[7] BIERMAN, G. M., MEIJER, E., AND SCHULTE, W. The essence of data access in C omega. In *ECOOP* (2005), A. P. Black, Ed., vol. 3586 of *Lecture Notes in Computer Science*, Springer, pp. 287–311.

[8] BIGGS, W. Plain Old Java Queries (POJQ). `https://pojq.dev.java.net`.

[9] BUEHRER, G., WEIDE, B. W., AND SIVILOTTI, P. A. G. Using parse tree validation to prevent SQL injection attacks. In *SEM* (2005), E. D. Nitto and A. L. Murphy, Eds., ACM, pp. 106–113.

[10] CAREY, M. J., AND DEWITT, D. J. Of Objects and Databases: A Decade of Turmoil. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1996), Morgan Kaufmann Publishers Inc., pp. 3–14.

[11] CODD, E. F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (1970), 377–387.

[12] Cω. `http://research.microsoft.com/Comega/`.

[13] COOK, W. R., AND IBRAHIM, A. H. Integrating Programming Languages & Databases: What's the problem? Submitted for publication may 2005, accessed November 2005 at `http://www.cs.utexas.edu/~/wcook/Drafts/2005/PLDBProblem.pdf`, 2005.

[14] COOK, W. R., AND ROSENBERGER, C. Native Queries for Persistent Objects. *Dr. Dobb's Journal* (February 2006). `http://www.ddj.com/documents/ddj0602e/`.

[15] COPELAND, G., AND MAIER, D. Making smalltalk a database system. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1984), ACM Press, pp. 316–325.

[16] Visual C# Developer Center. `http://msdn.microsoft.com/vcsharp/programming/language/`.

[17] IBM DB2. `http://www.ibm.com/db2`.

[18] db4objects. `http://www.db4o.com`.

[19] DIJKSTRA, E. W. Go to statement considered harmful. 351–355.

[20] Eclipse. `http://www.eclipse.org`.

[21] Eiffel. `http://www.eiffel.com`.

[22] Enterprise JavaBeans (EJB). `http://java.sun.com/products/ejb/`.

[23] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, March 1995.

[24] Hibernate. `http://www.hibernate.org`.

[25] JAMES GOSLING, BILL JOY, G. S., AND BRACHA, G. *The Java^{TM}Language Specification, 3rd Edition*. Addison Wesley Professional, 2005.

[26] Java programming language compiler (JavaC). `http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/javac.html`.

[27] JavaCC. `https://javacc.dev.java.net`.

[28] Java Data Objects (JDO). `http://java.sun.com/products/jdo/`.

[29] LEAVENS, G. T., AND CHEON, Y. Design by Contract with JML. Draft paper for Java Modeling Language, accessed May 2006 at `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`, 2006.

[30] LEAVITT, N. Whatever Happened to Object-Oriented Databases? *IEEE Computer 33*, 8 (2000), 16–19.

[31] LINDA DEMICHIEL, M. K. JSR 220: Enterprise JavaBeansTM, Version 3.0 - Java Persistence API. Tech. rep., Sun Microsystems, 2006. `http://jcp.org/en/jsr/detail?id=220`.

[32] The LINQ project. `http://msdn.microsoft.com/data/ref/linq/default.aspx?pull=/library/en-us/dndotnet/html/linqprojectovw.asp#linqprojec_topic5`.

[33] MCCLURE, R. A., AND KRÜGER, I. H. SQL DOM: compile time checking of dynamic SQL statements. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 88–96.

[34] Microsoft SQL Server. `http://www.microsoft.com/sql/default.mspx`.

[35] MØLLER, T., JENSEN, R. N., AND SÖNDER, P. Persistent Language Extension and Constructs for Java 1.5. Tech. rep., Faculty of Engineering and Science, Aalborg University, 2005.

[36] Oracle Corporation. `http://www.oracle.com`.

[37] PostgreSQL. `http://www.postgresql.org`.

[38] SALCIANU, A. D., AND RINARD, M. C. Purity and Side Effect Analysis for Java Programs. In *Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation, January 2005* (2005).

[39] SIPSER, M. *Introduction to the Theory of Computation*. PWS, Boston, MA, 1996.

[40] SQL-Java (SQLJ). `http://www.sqlj.org/`.

[41] Sun Microsystems. `http://www.sun.com`.

[42] TEAM, T. J. L. White Paper: About Microsoft's "Delegates". Tech. rep., JavaSoft, Sun Microsystems, Inc., 2001.

[43] Oracle TopLink. `http://www.oracle.com/technology/products/ias/toplink/`.

[44] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1987), ACM Press, pp. 227–242.

[45] WHALEY, J., AND RINARD, M. Compositional pointer and escape analysis for Java programs. *SIGPLAN Not. 34*, 10 (1999), 187–206.

A Signatures of the EntityCollection

```
public class EntityCollection<E> implements Collection<E> {

    // New method
    public EntityCollection(ResultSet rs) throws PersiJException;

    public EntityCollection();
    public static Connection getConnection() throws SQLException;

    // Methods implemented from java.util.Collection<E>
    public int size();

    public boolean isEmpty();

    public boolean contains(Object o);

    public Iterator<E> iterator();

    public Object[] toArray();

    public <T> T[] toArray(T[] arg0);

    public boolean add(E arg0);

    public boolean remove(Object arg0);

    public boolean containsAll(Collection<?> arg0);

    public boolean addAll(Collection<? extends E> arg0);

    public boolean removeAll(Collection<?> arg0);

    public boolean retainAll(Collection<?> arg0);

    public void clear();

    // Methods to manipulate data

    /*
     * Write all contained objects to database - updating and
     * inserting as necessary.
     */
    public void persist();

    /*
     * Update the representation of element in the persistant
     * storage.
     */
    public void persist(E element);

    /*
     * Deletes all elements from persistent storage, and clears the
     * underlying collection
     */
    public void unPersist();

    /*
     * Delete this element both from the collection and from persistent
     * storage
     */
    public void unPersist(E element);

    // Methods that affect query results

    /*
     * Set the largest number of objects that this EntityCollection may
     * contain
     **/
    public void setMaxSize(int maxSize);
}
```

B Grammar for SOQL

The grammar for SOQL can be found in Table B.1. It is on Extended Backus Naur Form (EBNF). Reading notes to the grammar: Elements in [] are optional (could also be written using ()?), a | between two elements is a choice of the two, elements marked with * are zero or more times, and finally elements marked with + are one or more times. Productions for IDENTIFIER, CHARACTER_LITERAL, and STRING_LITERAL, and parser-generator specific commands have been omitted for brevity.

```
CompilationUnit          ::=  MethodDeclaration
Identifier               ::=  <IDENTIFIER>
VariableDeclaratorId     ::=  <IDENTIFIER>
MethodDeclaration        ::=  [
                                 ( public
                                 | private
                                 | protected )
                              ]
                                 <EntityCollection> <Type>
                              | MethodDeclarator
                              | PersijMethodBlock
MethodDeclarator         ::=  <IDENTIFIER>FormalParameters
                                   throws PersiJException
PersijMethodBlock        ::=  {
                              EntityCollection<Type><IDENTIFIER>
                                    = new EntityCollection
                                         <Type>();
                              Type <IDENTIFIER> = null;
                              ( BlockStatement )*
                              return <IDENTIFIER>;
                              }
FormalParameters         ::=  ([FormalParameter
                                   (,FormalParameter)*])
FormalParameter          ::=  Type VariableDeclaratorId
Type                     ::=  ReferenceType | PrimitiveType
ReferenceType            ::=  ClassOrInterfaceType
ClassOrInterfaceType     ::=  Identifier
                                   [(.Identifier)*]
TypeArgument             ::=  ReferenceType
```

continued on next page

continued from previous page

PrimitiveType	::=	**boolean** | **char** | **int**
Name	::=	<IDENTIFIER>
		[(.<IDENTIFIER>)*]
Expression	::=	ConditionalOrExpression
		[AssignmentOperator
		Expression]
AssignmentOperator	::=	**=**
ConditionalOrExpression	::=	ConditionalAndExpression
		(||
		ConditionalAndExpression
		)*
ConditionalAndExpression	::=	EqualityExpression
		(**&&** EqualityExpression)*
EqualityExpression	::=	PrimaryExpression (
		(**==**|**!=**) PrimaryExpression
		)*
PrimaryExpression	::=	PrimaryPrefix
		[(PrimarySuffix)*]
PrimaryPrefix	::=	Literal
		| **(** Expression **)**
		| AllocationExpression
		| Name
PrimarySuffix	::=	.AllocationExpression
		| **.**<IDENTIFIER>
		| Arguments
Literal	::=	<INTEGER_LITERAL>
		| <CHARACTER_LITERAL>
		| <STRING_LITERAL>
		| BooleanLiteral
		| NullLiteral
BooleanLiteral	::=	**true** | **false**
NullLiteral	::=	**null**
Arguments	::=	**(** [ArgumentList] **)**
ArgumentList	::=	Expression (**,**Expression)*
AllocationExpression	::=	**new** PrimitiveType
		| **new** ClassOrInterfaceType
		Arguments
Statement	::=	Block
		| StatementExpression**;**
		| IfStatement
Block	::=	{ (BlockStatement)* }
BlockStatement	::=	Statement
StatementExpression	::=	PrimaryExpression
		[AssignmentOperator
		Expression]
IfStatement	::=	**if (**Expression**)** Statement
		[**else** Statement]

Table B.1: Grammar for SOQL.

Summary

This is a summary of this report. It is a mandatory part of a Master thesis in Computer Science written at Aalborg University, Denmark.

The broader underlying problem of this project, is that of the impedance mismatch encountered when trying to use relational databases in statically typed objects oriented programming languages. In a preliminary analysis, existing solutions to the impedance mismatch problem are reviewed, and are all found to be deficient in some manner. Posing the question: Why not change the premises? It is concluded that relational databases cannot be replaced by object-oriented databases to avoid the problem altogether. Reviewing existing work on what constitutes the impedance mismatch, a number of criteria are listed, that all must be fulfilled for a solution to properly solve the impedance mismatch problem.

Acknowledging that the scope of the project does not allow for the development of a full solution to the impedance mismatch problem, the focus of the project is narrowed down to solving one facet of the problem: Querying. Through an analysis of querying methods found in existing solutions to the impedance mismatch problem, a number of criteria are found to evaluate a querying method by: static checkable, automatic marshalling and unmarshalling, same paradigm as host language, minimal verbosity, minimal language alteration, modularizable, and optimizable.

Especially the approach Native Queries which promotes a way of querying where queries are expressed in the same paradigm as the host language (in this case object-oriented) is considered favorable, but still deficient due to a verbose fashion of implementation.

The specific problem addressed by this project is then defined as: *Is it possible to design an object-oriented query language integrated with Java, which can be transformed to Java code that ships the query as Structured Query Language (SQL) to the database while still fulfilling the criteria?*

In an effort to answer this question, a new language named Simple Object Query Language (SOQL) is designed. In order for the language to work, a rudimentary persistence framework named PersiJ is also described, based largely on Enterprise JavaBeans (EJB) version 3.

SOQL queries is basically methods in a Java program that have been marked with a special annotation. The allowed syntax inside these methods is a subset

of Java, and allows for primitives, classes, objects, messages (method calls) and branching (if-statements). Using these very basic building blocks SOQL is able to express predicates, branching of control flow, sorting, limiting and modularization of queries.

Investigating how the different elements of SOQL may be transformed using a number of informally defined partial transformation functions, a number of limits to the translation to SQL are found - especially in conjunction with method calls.

Finally, a discussion of how SOQL fulfills the criteria from the analysis finds that SOQL is statically checkable, has automatic marshalling and unmarshalling, minimal verbosity, is modularizable and optimizable. However, due to intricate differences in semantics from the host language Java, the criteria of being in the same paradigm as the host language is not considered to be fulfilled.

In conclusion SOQL is found to potentially be a favorable alternative to existing querying solutions, but while overcoming many important deficiencies simultaneously (being statically checkable while retaining modularization and optimization capabilities) the intricate differences of semantics between Java and SOQL are found to be a significant drawback.