2007

Language Integrated Query in Java for XML and Relational Database

Anurag Naidu

Language Integrated Query in Java for XML and Relational Database

Master's Thesis

Anurag Naidu

October 26, 2007

Committee

Dr. Axel T Schreiner, Chair

Dr. Rajendra K Raj, Reader

Dr. James M Kwon, Observer

Thesis/Dissertation Author Permission Statement

Title of thesis or dissertation: LANGUAGE INTEGRATED QUERY IN JAVA FOR XML AND RELATIONAL DATABASE

Name of author: ANURAG NAIDU
Degree: MS. COMPUTER SCIENCE
Program: COMPUTER SCIENCE
College: GCCIS

I understand that I must submit a print copy of my thesis or dissertation to the RIT Archives, per current RIT guidelines for the completion of my degree. I hereby grant to the Rochester Institute of Technology and its agents the non-exclusive license to archive and make accessible my thesis or dissertation in whole or in part in all forms of media in perpetuity. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Print Reproduction Permission Granted:

I, Anurag Naidu, hereby **grant permission** to the Rochester Institute Technology to reproduce my print thesis or dissertation in whole or in part. Any reproduction will not be for commercial use or profit.

Signature of Author: Anurag Naidu Date: _____

Print Reproduction Permission Denied:

I, _____, hereby **deny permission** to the RIT Library of the Rochester Institute of Technology to reproduce my print thesis or dissertation in whole or in part.

Signature of Author: _____ Date: _____

Inclusion in the RIT Digital Media Library Electronic Thesis & Dissertation (ETD) Archive

I, Anurag Naidu, additionally grant to the Rochester Institute of Technology Digital Media Library (RIT DML) the non-exclusive license to archive and provide electronic access to my thesis or dissertation in whole or in part in all forms of media in perpetuity.

I understand that my work, in addition to its bibliographic record and abstract, will be available to the world-wide community of scholars and researchers through the RIT DML. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation. I am aware that the Rochester Institute of Technology does not require registration of copyright for ETDs.

I hereby certify that, if appropriate, I have obtained and attached written permission statements from the owners of each third party copyrighted matter to be included in my thesis or dissertation. I certify that the version I submitted is the same as that approved by my committee.

Signature of Author: Anurag Naidu Date: _____

ROCHESTER INSTITUTE OF TECHNOLOGY

The Undersigned Computer Faculty Approves the Thesis

Language Integrated Query in Java for XML and Relational Database

Axel T. Schreiner

Dr. Axel T Schreiner, Chair

Rajendra K. Raj

Dr. Rajendra K Raj, Reader

James M. Kwon

Dr. James M Kwon, Observer

Thursday, 25 October 2007

10 - 11 AM, 70-3000

Abstract

XML and relational databases are the most commonly used data-sources for numerous Java–based enterprise applications. The ever–growing dependence of Java–based applications on these technologies calls for developing a uniform means of querying these data–sources in the Java layer such that queries written for any of these technologies look similar and are in fact governed by a single grammar. This thesis implements a new backend system encapsulating querying capabilities over XML and JDBC based relational databases which would complement the translation of queries written using QuEL [1], a language extension to the Java programming language. The QuEL language extension to Java is part of another thesis developed previously and acts as the query language for the back–end implemented in this thesis for XML and relational databases. The benefits of this approach include faster development time and a flat learning curve towards a single query language for various data sources.

Contents

1 Introduction

Query languages are programming languages that are used typically to retrieve data from databases or other information storage mediums. The growing dependence of major applications on a variety of database systems and other mediums of storage, like XML [2], now makes it important that we are able to query such data–sources in an efficient and consistent manner. These various data storage technologies have evolved over a long period of time and originated from separate companies or special interest groups. What this means is that each of these different technologies has its own distinct languages which are used to write queries for retrieving data.

In this thesis we aim to address the problem of heterogeneity in querying different data–sources with an aim to introduce a syntactically consistent query mechanism that is almost independent of the nature of the data storage medium that is queried. We demonstrate the various aspects of query preprocessing that need to be changed to make the consistent query experience integrate into Java.

The .NET community at Microsoft Corporation has made considerable progress in developing Language Integrated Query (LINQ) [3] for the C# and VB language specification in .NET 3.0. This thesis work in conjunction with the thesis "Integrating a Universal Querying Mechanism in Java" [1] is targeted towards developing a back-end for such a query language that would provide similar benefits in Java for retrieving data out of XML documents and relational databases. One could now write queries using QuEL [1], as a language extension of Java, just as one would write SQL [4] queries on a database console or XPath [5] queries to operate on a XML document. A possible future extension of both the theses would be to incorporate QuEL [1] into the Java grammar which would make

for a powerful feature in Java that could be used to query XML and relational databases just like one would query on Java objects.

2 Core concepts

This chapter takes a closer look at the design and architecture of the two data–sources that will be the target for QuEL queries for the purpose of this thesis. Simple examples will be used to demonstrate how writing queries in SQL [4] and XPath [5] works by using the existing APIs provided in Java and what are the concerns with writing such queries. This would help in understanding both the advantages of writing queries using QuEL [1] as well as the backend designed in this thesis for XML and relational database.

2.1 Hierarchical Data Model / Extensible Markup Language (XML)

XML is a specification for defining markup languages designed to separate data from its representation. This makes it a very useful format for data representation and data interchange between different systems. Though relatively new in the computer industry, XML has become very popular and is widely used as an information storage medium and configuration data–source in software applications.

Consider beginning with a very simple XML document that has information for the staff in an organization, to see how a document like this looks and understand the structure of such a document. The document has first name, last name and phone number information of each staff member of the department and maintains them as an XML document as depicted in the Figure 1 below.

```
<staff>
   <person>
      <first>Sandy</first>
      <last>Ferrara</last>
      <phone>55178</phone>
   </person>
   <person>
      <first>Tina</first>
      <last>Sturgis</last>
      <phone>52529</phone>
   </person>
   <person male='true'>
      <first>Jason</first>
      <last>Harrison</last>
      <phone>52529</phone>
   </person>
</staff>
```

Figure 1: `Staff` XML document as a tree

A quick look at the document shows a striking resemblance to a tree structure which is rooted at the staff node.

Given this tree-like visualization of the document one could imagine querying this document being analogous to traversing the tree and pruning out the leaves that do not pass the filtering criteria. One would have to write many lines of code in Java using the W3C DOM [6] to read the XML document into memory and then write a visitor to collect nodes that pass the filtering criteria of the user query and construct a new document with reduced nodes. Although this is a completely solvable problem, it is also an extremely common

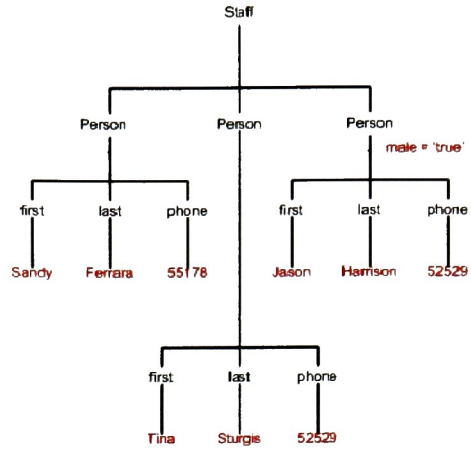


Figure 2: `Staff` XML document as a tree

scenario in the programming domain to sieve through XML documents and collect data of interest at particular points.

XPath [5] is one such query language tailored to operate efficiently over XML documents to filter out data. XPath queries are crisp, non-verbose and extremely powerful tools for working with XML. Since XML has been constantly gaining importance in the software industry, a marriage of some sort between XPath and Java was always a logical next step. The growing adoption of this technology in the industry was cemented further when Sun introduced XPath APIs in Java 5 (JDK 1.5) for querying XML documents, which made available the power of efficient XPath querying from within user-level Java code. XQuery [7] is also one such mechanism that can be used to query an XML document.

Using following two simple XML documents, `Rooms` and `Staff`, how a query can be written using XPath [5] and the Java XPath APIs to extract data from these files.

Figure 3: XML document for `Rooms` and `Staff`

`Staff` is the same file that we saw above and `Rooms` is a file that maps the room number with the phone number associated with that room. A querying scenario from these two documents could be to search across the files and match records on phone numbers across the documents and return staff names with the corresponding room they occupy. Fairly common as this scenario may be, it is not exactly a programming cakewalk to achieve the desired result.

Assuming that the developer has good knowledge of XPath, XSLT [8] and HTML, the code for obtaining the results of the query across the two XML files would look something like this in a XSLT file

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes" />
<!-- Read in all the room nodes from the Rooms xml file into a temporary
      variable-->
<xsl:variable name="data" select="document('rooms.xml')/rooms" />
<!-- Match the root node -->
<xsl:template match="/">
```

```
<!-- Construct the html for the page headers etc.  -->
<html>
   <head><title>Occupancy page</title></head>
   <body>
   <table border="1" align="center"> <tr><td colspan="3" align="center">

      <b>Name and Room occupancy</b></td></tr>
      <xsl:apply-templates select="/staff" />
   </table>
   </body>
</html>
</xsl:template>
<!-- Declare the template for matching the staff and room -->
<xsl:template match="staff">
<!-- Apply the template over all the staff nodes -->
<xsl:for-each select="./child::*">
   <xsl:variable name="p" select="./phone/text()" />
   <xsl:variable name="f" select="./first/text()" />
   <!-- Get all the rooms to test and iterate -->
   <xsl:for-each select="$data/child::*">
      <xsl:if test="./phone/text() = $p">
         <tr>
         <td><xsl:value-of select="$f"/></td>
         <td><xsl:value-of select="./@room"/></td>
      </tr>
   </xsl:if>
</xsl:for-each>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Figure 4: XSLT to merge `Staff` and `Rooms`

The result of the above XSLT on the two XML documents yields the following HTML output as can be seen in Figure 5.

Notice the nested loops `xsl:for-each select` with XPath expressions that do the trick along with the presentation HTML code that displays the result. One could debate the complexity of the code, however there is no doubt that its excessively wordy and cumber-

Name and Room occupancy	
Sandy	3021
Tina	3012
Tina	3671
Joanne	3008
Eileen	3005
Jason	3005
Christina	3022
Liane	3022
James	3599
Sam	3596

Figure 5: Result of XSLT solution

some to write, given how common this scenario is when working with XML documents. What we have looked at is a non-Java implementation of the query. Reconsider how the same can be achieved using Java and the Java XPath APIs.

```
public class XmlSample {

  /**
  * @param args
  */
  public static void main(String[] args){

     DocumentBuilder documentBuilder = null;
     Element staff = null;
     Element rooms = null;
     XPath xpath = null;
     String staffXPathExpr = "/staff/person";
     String roomXPathExpr = "/rooms/room";
     NodeList outer = null;
     NodeList inner = null;
     try {
        documentBuilder =DocumentBuilderFactory.newInstance()
           .newDocumentBuilder();
     } catch (ParserConfigurationException e) {
        e.printStackTrace();
```

```
    }
try {
      staff = documentBuilder.parse(new File("./linq/staff.xml"))
         .getDocumentElement();
      rooms = documentBuilder.parse(new File("./linq/rooms.xml"))
         .getDocumentElement;
   } catch (SAXException e) {
      e.printStackTrace();
   } catch (IOException e) {
      e.printStackTrace();
   }

   xpath = XPathFactory.newInstance().newXPath();

   try {
      outer = (NodeList) xpath.
         evaluate (staffXPathExpr, staff, XPathConstants.NODESET);
      inner = (NodeList) xpath.
         evaluate(roomXPathExpr, rooms,XPathConstants.NODESET);
   } catch (XPathExpressionException e) {
      e.printStackTrace;
   }

   // Match the results in nested loop
   // THIS IS THE ACTUAL QUERY LOGIC
   int staffCount = outer.getLength();
   int roomCount = inner.getLength();
   String roomNum = "";
   Node roomItem = null;
   for (int i = 0; i < staffCount; i++) {
      Node staffItem = outer.item(i);
      try {
      roomNum = xpath.evaluate("phone/text()", staffItem);
         for (int j = 0; j < roomCount; j++) {

            roomItem = inner.item(j);
         if(roomNum.equals(xpath.evaluate("phone/text()", roomItem)))
         System.out.println(xpath.evaluate("first/text()", staffItem)
         + " "+ xpath.evaluate("@room", roomItem));
         }
      } catch (XPathExpressionException e) {
```

```
            // TODO Auto-generated catch block
               e.printStackTrace();
            }
         }
      }

      }
```

Figure 6: Java code for XML merging

The first thing that catches the eye is the wordiness of the solution and though it solves the problem, maintainability of repeated boilerplate code, mixed with the actual functional logic will always remain an issue over time. This problem is especially of concern as such queryies for XML documents are fairly commonplace.

Let us elucidate the problems seen so far in writing queries, given the existing infrastructures and technologies, which would help explain why a change in this area is long overdue.

- Complex code logic needed for fairly simple query patterns.
- Proficiency in many technologies like XPath, XSLT, JDK XPath APIs etc has to be combined.
- Repetitive boilerplate code to supplement the actual query code.
- The query itself remains type unchecked and so potentially unsafe.
- Lack of consistent query language for all data-sources.

We will come back to these observations when we discuss the design for QuEL and explain the workings of the XML backend for the new language.s

2.2 Relational Data Model / Relational Database (SQL)

A relational database is a data-store that is based on a relational model between entities, a model that works on relational algebra and set theory. Such a relational model was first proposed by Edgar Codd [9] where the word `relation` refers to a table in the database.

The smallest building block of a relational database is called an attribute, which is a name/value pair. An unordered set of attributes forms a tuple and an unordered set of such tuples makes up a relation, which lends its name to a relational database. However when dealing with relational databases we use the words table, row, column and constraints which are equivalent terms that map the mathematical concepts on which relational databases were designed to an actual database implementation.

Some older databases were also designed on a hierarchical data model like that of XML. Most modern databases have moved into a relational model because of the greater flexibility and advantages that it provides. One of the specific advantages with a relational model is a many-to-many relations between entities. One of the severely limiting conditions for a hierarchical model is that there can at best be a one-to-many relational mapping between entities; however, the model cannot accommodate many-to-many relations.

A simple example of this could be a super-store where products are classified based on the various sections like apparel, grocery, sports goods etc. The products in the display line are classified under only one section, however if there are products that really fall into more than one section, then the database does not provide any option to list the same product as belonging to more than one section. This is a serious limitation in terms of defining the most accurate classification of products and performing reverse lookup on products in

order to gather accounting information from sections for sales statistics. This discussion is merely to provide an insight into why the database industry has moved towards a relational model for its database needs.

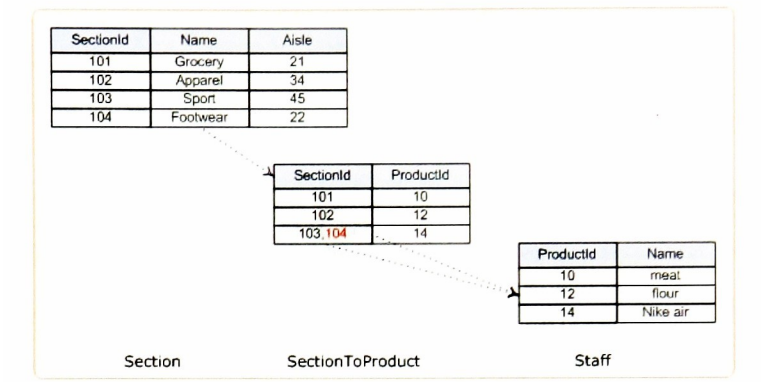
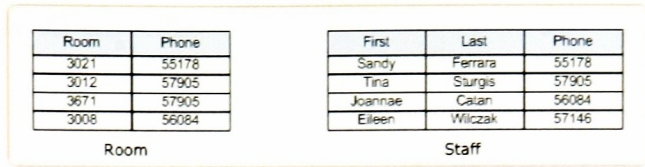


Figure 7: Hierarchical model anomaly

We discussed the obvious advantage of relational databases and how the database industry has moved towards a relational model, focus now on the technologies that one needs to familiarize oneself with in order to be able to work with relational models. The primary means of accessing relational databases is via the Structured Query Language (SQL) [7] invented by IBM in the 1970s.

In order to see how SQL is used consider the following two relational database tables `Staff` and `Rooms` that contain the same data as shown earlier but this time in relational data model. In order to perform the same query as we did for the XML sample we would now write a SQL query that joins the two `Staff` and `Rooms` tables on the matching phone number column and gives a result mapping the staff name with the room they occupy.



Room	Phone
3021	55178
3012	57905
3671	57905
3008	56084

Room

First	Last	Phone
Sandy	Ferrara	55178
Tina	Sturgis	57905
Joannae	Calan	56084
Eileen	Wilczak	57146

Staff

Figure 8: `Room` and `Staff` tables

The following SQL query returns the result as listed below. The SQL query itself is very intuitive and simple in structure and does the job, however database are always coupled with some application for which they serve as the data-store and thus the measure of the ease of use of a database in the manner in which it may be accessed by the application it serves.

```
SELECT S.FIRST, R.ROOM
FROM STAFF AS S JOIN ROOM AS R
ON S.PHONE = R.PHONE
```

First	Room
Sandy	3021
Tina	3012
Joannae	3008

Figure 9: SQL query and result

Java has the necessary APIs for writing SQL queries in user level Java application code. These APIs were introduced into the language a long time ago and have evolved since then in terms of the functionality they provide. The same SQL query can be written within a

Java program again with a large amount of code wrapped around the actual query. This addition setup code, though completely functional and correct, has similar drawbacks as outlined for the Java XPath APIs.

```
public static void main(String[] args) {
   // Connection setup
   Connection con = null;
   try {
      Class.forName("org.hsql.jdbcDriver");
      con = DriverManager.getConnection("localdb", "admin", "password");
   } catch (ClassNotFoundException e) {
      e.printStackTrace();
   } catch (SQLException e) {
      e.printStackTrace();
   }

   try {
      PreparedStatement ps = con
         .prepareStatement("SELECT S.FIRST, R.ROOM FROM STAFF AS S
               JOIN ROOM AS R ON S.PHONE = R.PHONE;");
   ResultSet rs = ps.executeQuery();
   while (rs.next())
      System.out.println("NAME: " + rs.getString(0) + " , ROOM: "
         + rs.getInt(1));
   } catch (SQLException e) {
      e.printStackTrace();
   }
}
```

Figure 10: Java code using JDBC to access database

The power and speed of SQL is immense but with no type–safety and the requirement for boilerplate wrapper code with its usage in Java creates a scope for further improvement.

3 Language Integrated Query (LINQ)

The previous chapter introduced two prominent data-stores that are commonly used with enterprise level applications and the existing technologies available to query them. The areas of concern highlighted while using these technologies points a direction towards an approach aimed at alleviating those concerns and making the entire query experience as homogeneous as possible with additional safety checks that make it consistent with the high level programming paradigm.

Language Integrated Query [10] is a project at Microsoft that is picking up momentum in the programming language community for its attempt at bringing in a querying syntax into mainstream programming languages. The LINQ mechanism provides the programmer with a way to programmatically organize and extract specific information from a data source. It provides a consistent syntax for writing queries over various data-sources like object collections, XML, relational databases and even third party data-sources as long as some of the contract requirements are implemented. Currently in LINQ a data source can be a hierarchical document, like XML, a database, like mySQL, the object collection, or any collection that implements the Enumerable interface. Given the type of one of these data sources, the user has the ability to extract information from this source using the same words in the host language. From this vantage point the programmer is extracting information in the same manner from three entirely different sources.

The .NET platform and C# 3.0 language in particular has incorporated useful language features that makes seamlessly incorporating LINQ into the .NET platform feasible. These include Anonymous Classes, Extension Methods [11], Lambda expressions etc. All these languages features together allow LINQ exist. The LINQ mechanism takes a set of re-

served words in the language and maps these words into a series of method calls on the data–source. These method calls are mapped to certain logical operations. For instance, the logic of a join operation is mapped to a method named `join()`. This direct mapping allows for many by products that do not exist in traditional queries performed on data sources such as databases and hierarchical documents.

One of the biggest results of this technology is type–safe queries, ensuring at compile time and not runtime, that these queries are valid and will not fail. Ensuring proper typing reduces the number of potential bugs in a program in the same way a strongly typed language like Java removes errors that plague a not so strongly typed language like C or C++.

Consider the query below in LINQ over an in-memory collection of `Employee` objects to find the names of employees that are of the same age.

```
... var result = from e1 in Employee
    from e2 in Employee
    where e1.age() == e2.age() && e1.name() != e2.name()
    select new {person1 = e1.name() ,person2 = e2.name()};
```

Notice how the query is simple in its structure and no nested `for` loops are needed as would be required in a traditional program. The `from` keywords brings in new data sources into the query environment, the `where` keyword does the comparison (like in an `if` statement) and the `select` keyword constructs the return type of the result. The new object with the names of employees is not previously defined but it is constructed at runtime and assigned to a dynamically typed result `var`.

In the later section for QuEL [1] we shall see how we get around the language features in

C#, like Anonymous Classes, Extension Methods [11], Lambda expressions that are not part of Java, in order to implement QuEL for XML and relational databases.

3.1 XLinq and DLinq

The runtime for LINQ on different data–sources differs in ways that are specific to handling the respective data-sources though from a user's perspective LINQ offers the idea of a consistent query experience across any data–source backend.

3.1.1 XLinq

The runtime that provides Linq capabilities for XML querying is just an extension of Linq for data objects, with some special extension methods that are relevant in the context of reading XML documents, like the `elements()` method to read the text value of nodes and `attrib()` method used to read the text value of an attribute of a node. The rest of the Linq grammar stays the same, An XLinq query is thus structured exactly the same way as a Linq query on collections of data objects.

3.1.2 DLinq

The Linq grammar by design resembles the syntax of SQL closely. Thus no additional grammar tokens are needed for extending the concept of Linq to databases. The implementation for SQL remains the same. Extension methods in the host language backend corresponding to keywords in the query language perform the actions required to fulfill the semantic meaning of that keyword in the SQL grammar.

4 QuEL

We have looked at the problems identified with query languages and their integration into a host high level programming language. We also discussed how the LINQ project at Microsoft is bringing together query syntax inside mainstream application programming languages like VB.NET and C#. In this chapter we will look at the recently offered potential solution for query language support within the Java programming language, called QuEL [1]. The offered solution is a front-end that constitutes the new QuEL grammar and the translation mechanism that is so far implemented for performing queries over object collections.

The QuEL (Query Enhanced Language) mechanism is a front-end preprocessor which when given a Java file with an embedded query, written using the QuEL grammar, will convert that language into syntactically correct Java code, thereby making queries on Object collections less cumbersome and allowing the programmer to ignore the boiler plate coding normally needed while performing these queries with the supported Java APIs so far available.

4.1 QuEL front–end

The front–end for QuEL as developed in the thesis "Integrating a Universal Querying Mechanism in Java" [1] consists of the complete QuEL grammar and the translation mechanism that parses a query written in QuEL and performs various preprocessing steps needed for it to map the query to the appropriate back–end. The steps in this proprocessing involves converting the parsed tokens of a query into an internal *method tree* representation. This

method tree representation is an abstract mapping of the query to any back–end implementation. The next step is performed by an interpreter that converts this abstract mapping to a custom mapping in the form of pure Java code.

4.2 Object Collections

So far the interpreter of the QuEL front–end is configured to use the back–end written by Dr. Axel Schreiner for the purpose of querying on object collectios [12] in Java. The Java code emitted by the interpreter makes method calls defined in this back–end as a successive method call chain. These methods are the real operators on the object collections on which the original QuEL query is written, thereby returning the result as dynamically computed typed object collection which is again usable in user written Java code. One could imagine the back–end as the proxy for executing the actual query and the query language as just syntactic sugar for work done by the back–end.

The same LINQ query from the previous chapter is now slightly modified and is now the QuEL query imbedded in Java code within the `@@` markers. This query does the exact same function as described earlier but now in the Java enviornment using the QuEL front–end and object collection back–end that was discussed.

```
... @@
result = from edu.rit.cs.quel.Person e1 in Employee
      from edu.rit.cs.quel.Person e2 in Employee
      where e1.age() == e2.age() && e1.name() != e2.name()
      select new {person1 = e1.name() ,person2 = e2.name()}
@@ ...
```

5 QuEL for XML and Relational Database

We discussed in the previous chapter that the QuEL [1] mechanism is a front-end preprocessor which takes in a Java file with a QuEL query and translates to pure Java code for object collections. The promise of this paradigm is however not just to provide a query mechanism for data collections but a consistent querying experience across varied data sources. The work done in this thesis is aimed at providing just that experience, for which we have chosen XML and relational database as the other data sources for which the query mechanism can be extended.

In an earlier chapter on core concepts, it was discussed that writing query over XML or a relational database inside of Java code is cumbersome and involves writing a lot of code that a programmer, for the most part, does not like to be bothered about. It can be debated that the programmer should not be expected to care about all the wrapper code and focus around the actual logic of query. This is one of the basic reasons behind developing QuEL and extending it to all data–sources like XML and relational database so that the focus of querying inside of Java comes back to the query logic.

As discussed in the previous chapter, the grammar for the new language QuEL has been developed under the thesis "Integrating a Universal Querying Mechanism in Java" [1] where QuEL will be used to write queries on collections inside a Java program that would be preprocessed by the QuEL compiler to transform the QuEL part of the code into pure Java syntax which could then be compiled by `javac`.

The current thesis work utilizes the QuEL grammar for writing queries over DOM and SQL constructs. The QuEL grammar completely incorporates the SQL syntax constructs,

however some constructs specific to the DOM implementation in Java is not part of the grammar. Thus for the purpose of this thesis the back–end for XML assumes the recognistion of these constructs in the grammar so that pure Java code could be build from QuEL queries for XML data–source. Like in the QuEL paper [1] the approach taken to integrate these queries is a preprocessor based approach, so that the extensibility of the Java programming language can be easily proven. This approach also gives us the software engineering advantage of maintainability and independence from subsequent Sun JDK releases. The processes of incorporating the new grammar into Java would then be to merge the QuEL grammar and the elements of our new compiler and the query back-end support into the Java compiler.

Figure 11 provides a high level understanding of the process that converts the special Java file with QuEL query through a pre-processing stage into a complete compile able Java file with the right mappings generated to serve the query functionality.

The process of compilation of QuEL injected Java code is a two stage process where Java file is first pre-processed by the QuEL processor that parses only the QuEL code within "@@" markers. The lexer for the preprocessor converts the query into a parse tree that is then mapped to a method tree. The methods in this tree are the methods that correspond to each keyword of the query grammar like `from`, `where`, `select`, `join` etc. The corresponding methods are members of DocumentFragment or Queryable interface depending on the data source. The methods form a closed set of operations over the resulting type. Since the operations are closed on the result it makes it very simple to apply just about any action on the result at any stage.

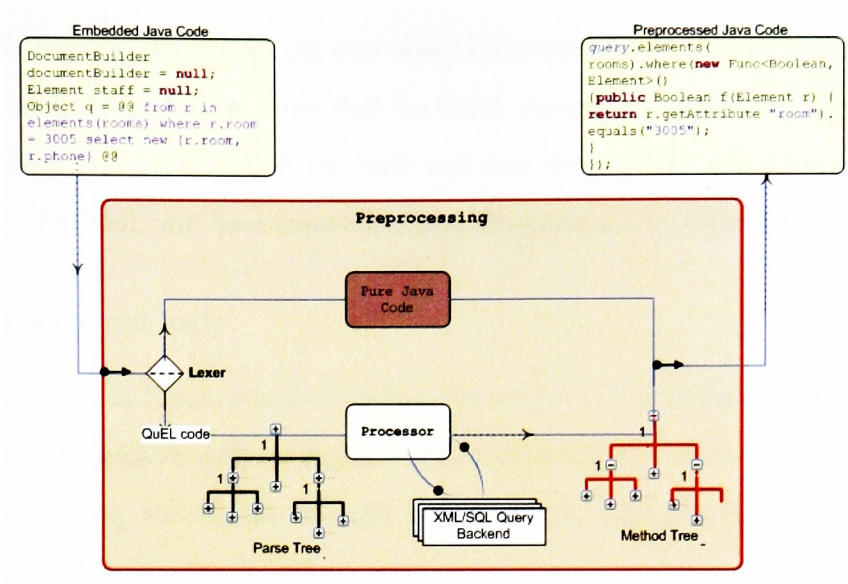


Figure 11: QuEL preprocessing process

Lazy evaluation

The design of any backend implementing the required interface actions has to be made a closed set in order to achieve a cascaded evaluation flow which is also a "lazy" evaluation mechanism reminiscent in the functional programming language Haskell. The evaluation chain is triggered only when result of the query is first used and not while each part of the query operation is constructed. This gives the flexibility of "adding on" any of the closed set of operations on an existing query, and as a side effect this works well for writing nested queries.

5.1 XML in QuEL

Now c[illegible]r how we can extend the new query framework to be able to write queries over XML documents. Also keep in mind that the QuEL grammar is generic for a uniform query syntax however because internally the back–end uses the `org.w3c.dom` APIs in Java, parts of the query for XML will have constructs that are governed by these APIs.

5.1.1 Background logic

The Java program in Figure 6 is the baseline that we will use to compare the improvements that we claim to achieve with an implementation of QuEL for XML. Here is all that now needs to be written with QuEL in order to read in the Staff and Rooms document and merge them to generate the occupancy list.

```
   final Element staff = documentBuilder.parse(
      Tester.class.getClassLoader().getResourceAsStream(
      "quel/staff.xml")).getDocumentElement();
   final Element rooms = documentBuilder.parse(
      Tester.class.getClassLoader().getResourceAsStream(
         "quel/rooms.xml")).getDocumentElement();
@@
   result = from Element s in elements(staff)
         join Element r in elements(rooms) on
            s.getElementsByTagName("phone") matches
            r.getElementsByTagName("phone") into sr
         select new sr.first, sr.room
@@
```

Figure 12: QuEL query for XML access

Notice how you just have to read in the *Staff* and *Rooms* documents once into the DOM

before you start the query. The query then is a very compact statement that uses the `from` keyword to introduce the *Staff* source and the `join` keyword is used to introduce the *Rooms* source and match on the *phone* node on both the documents that have the same value and merge the results of the `join` filter into a temporary structure referred by the variable `sr`. Now the `select` keyword is used choose the nodes/attributes from the merged document into a final result which is a DOM `DocumentFragment` structure.

5.1.2 Method translation

In order to understand the workings behind the scene that makes the whole query look as simple, less wordy and yet achieve the same result, we will start by looking at the architectural abstracted view of the processing that is part of the pre-processing stage of the compilation.

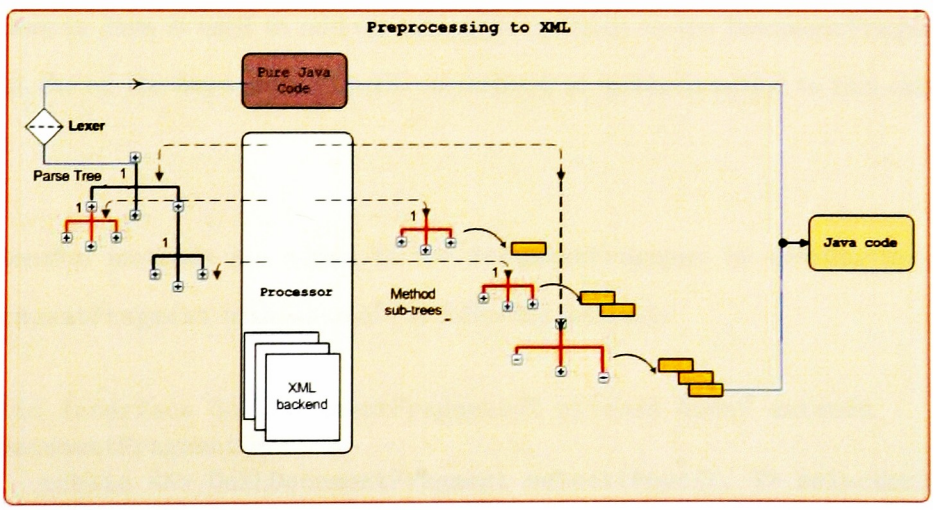


Figure 13: XML query translation

From Figure 13 it can be seen that a Java file with QuEL code is parsed by the lexer and a parse tree of with the nodes of the query is internally generated. Note that the pure Java code part of the file is not scanned in this process and passed though this pipeline without any modification. After the parsing is done a mapper(not shown in figure) walks the parse tree and converts the tree into something called a `Method tree`. This tree is closer representation of the final Java output that will get generated and has all typing related information along with the original query lambdas and predicates that can now be worked on by the interpreter that converts it to the final Java code that can be compiled with `javac`.

5.1.3 XML backend design

In this section we will describe the workings of the backend designed for XML and how the Proxy class in Java is used to add on extension methods to the `DocumentFragment` class such that the all the keywords in QuEL correspond to methods added to this class during runtime.

The extension methods are added to the `DocumentFragment` by defining an interface `QuELDocumentFragment` with some of the following methods:

```
public interface QuELDocumentFragment<T extends Node> extends
   DocumentFragment {
      public <K> QuELDocumentFragment select(Func<K, T> selector);
      public <K> QuELDocumentFragment groupBy(Func<K, T> group);
      public <K> QuELDocumentFragment orderBy(Func<K, T> group);
      public <T extends Node> join (QuELDocumentFragment<Node> joinSource,

      .
```

```
            Func<Boolean, Node> predicate1, Func<Boolean, Node> predicate2,

            Func2<Node, Node, Node>selector)
}
```

The first step in the query interpretation process is to create an instance of the Query class, which is the entry point for creating a proxy of the DocumentFragment class. This is done by making a call to the elements() method defined in this class with the following signature:

```
public <T extends Node> QuELDocumentFragment elements(Node container)
```

This method takes in an XML document read in as a Node object and wraps a `QuELDocumentFragment` around it. In addition it also performs the check `p.getNodeType() == Node.ELEMENT_NODE` on each node of the input document before it is returned to the next method in the query composition chain. This is important because when a XML document is read into the DOM the nodes constitutes the following sub interfaces like `Attr, Comment, Document, Entity, Element, Text` etc and for the purpose of the query we will for all practical purposes be concerned only with the `Element` node. This way the `elements()` is not only used to introduce the proxy wrapper but also to initialize the basic filtering for the query being constructed.

Once the proxy is introduced all the QuEL keyword call method definitions are now visible and can be invoked at any point on the resulting object of each method since the return type for each of these methods is the same type on which the method was invoked, namely in the XML case a `QuELDocumentFragment` class. Here is how a proxy object of the same type is returned from each of the extension methods:

```
return (QuELDocumentFragment) Proxy.newProxyInstance(
QuELDocumentFragment.class.getClassLoader(), new Class[] {
   QuELDocumentFragment.class, DocumentFragment.class },
      new QuELDocumentFragmentHandler(this.db, values,
         new Evaluate()
         {
            public DocumentFragment eval() { }
            ...
            ...
         }
      )
   );
```

It's the body of the `Evaluate` interface that has the `eval()` method that is distinct for each of the extension methods that have logic specific to the action that needs to be performed by that method.

Consider a query in QuEL and see the converted Java code for the same using the backend design for XML.

```
@@
   result = from Element s in elements(staff)
      join Element r in elements(rooms) on
         s.getElementsByTagName("phone") matches
         r.getElementsByTagName("phone") into sr
      where s.getElementsByTagName("last").item(0)
         .getTextContent().equals("Catan")
      select new sr.first, sr.room
@@
```

Figure 14: Another QuEL query for XML access

The preprocessing logic would internally convert the document into `DocumentFragment`

object(s) that would have special DOM like extension methods that would be used to extract node elements and attributes from the DOM. The query above in QuEL would look like the following when mapped to the Java backend designed in this thesis.

```
QuELDocumentFragment l = query.elements(staff).join(
   query.elements(rooms), new Func<String, Element>() {
   public String f(Element s) {
         return s.getElementsByTagName("phone").item(0)
         .getTextContent();
      }
   , new Func¡String, Element¿()
      public String f(Element r) {
      return r.getElementsByTagName("phone").item(0)
         .getTextContent();
   }
   , new Func2¡DocumentFragment, Node, Node¿()
   public DocumentFragment f(Node s, Node sr) {
      return query.combine("s", s, "sr", sr);
   }
   ).where(new Func¡Boolean, Element¿()
      public Boolean f(Element s) {
         return s.getElementsByTagName("last").item(0)
            .getTextContent().equals("Catan");
}
});
```

Figure 15: Java translation for QuEL query for XML

The code generated using the XML backend Query class can now be compiled using javac would generate the same result as the Java code that we wrote in *Core Concepts* chapter 2.

Although the generated Java code is type safe and compiles, the query itself can still fail to generate the expected result. The reason for that is, XML is not schema enforceable and thus query itself cannot be type checked for correctness against the XML document that it would work against. Schema aware queries [13] is an idea developed by Dr. Michael

Kay and aims at providing type checking on queries written using XPath. This is howwever an optional facility and Java at this time does not support it, so we ignore this aspect for the scope of this work.

Thus a possible future extension to the QuEL for XML could be to makes the translated queries completely type safe for documents that adhere to some schema definition. However for all other XML documents it would be a very challenging task to do achieve complete query type safety and is thus kept outside scope of this thesis.s

5.2 Database query in QuEL

Now considering how same query translation mechanism works for a new data source, in this case we will look at the most commonly uses database medium, Relational databases. The big challenge and difference while tackling this problem is, that unlike in the previous cases for object collections or XML, the data source is on disk and should always remain on disk during the entire query process. This might sound trivial at the moment but it is an important situation that needs to be maintained and which influences the design on the backend for SQL, i.e. since all the information cannot be cached we have to translate the query into SQL. We shall discuss that later in this chapter after we have built the foundation for query translation.

The other important aspect of query mapping for relational databases is that of type checking. The database tables over which the QuEL query executes exists on disk and because the Java compiler has no way prior knowledge of their types, some way needs to be devised to acquire typing information about such entities so that we can provide a type-safe query guaranteed to execute successfully.

5.2.1 Background logic

Once again reconsider the Java program in Figure 10 as the baseline to compare with for the improvements that we claim to achieve with an implementation of QuEL for relational database. Here is all that now needs to be written with QuEL in order to read in the Staff and Rooms tables and merge them based on the phone number field to generate the occupancy list.

```
final edu.rit.cs.quel.StaffBean staff = new
      edu.rit.cs.quel.StaffBean(Staff);
final edu.rit.cs.quel.RoomBean rooms = new
      edu.rit.cs.quel.RoomBean(Rooms);
@@
result = from edu.rit.cs.quel.StaffBean s in staff
   join edu.rit.cs.quel.RoomBean r in rooms on
   s.phone matches r.phone
      where r.size > 2000
      select new s.first, r.size
@@
```

Figure 16: QuEL query for database access

Notice that the query is a very compact statement that uses the from keyword to introduce the Staff source and the join keyword is used to introduce the Rooms source and match on the phone column on both the tables that have the value and on the result apply the filter condition where the size of the room is greater than 2000. The select keyword is used to project the columns from the filtered temporary result into a final result which is a `Queryable<T>` type, where `T` is the return type of the query which can be an internally generated type or any of the standard built–in types.

5.2.2 Method translation

Figure 17 provides an understanding of the workings of the translation system for QuEL for SQL and the corresponding SQL backend.

The process of converting QuEL for relational database query is slightly different from the process discussed earlier for XML. The hook for conversion of the QuEL processor parse tree into the appropriate method call tree on the SQL backend plugs in earlier in the

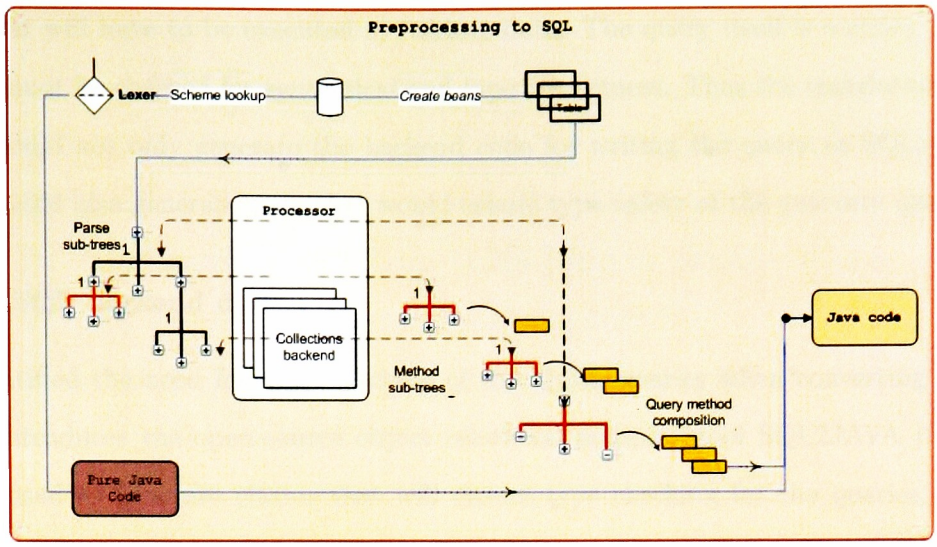


Figure 17: SQL processing

whole translation scheme. The reason for this difference is for the following factors:

- QuEL queries for relational databases are converted to their SQL equivalent query before they get executed on the database. This approach was taken in order to leverage the performance that SQL provides for querying databases, especially in cases when the scalability of the query translation mechanism is considered then executing the appropriate SQL query to extract data becomes a primary concern for designing the backend. Another approach that was considered initially, involved frontloading the data in all the tables into data objects called Beans. Once the all the data in loaded the query could then be run in–memory on the beans, like in the object collection backend [12]. This approach works well for objects however the scale of database tables is far larger and it cannot be assumed that the system has enough memory resources to store all the data in–memory. Also it would not be a good design approach that stretches the limits of the system resources.

- The other aspect of this design is to convert the query into the SQL query string

that will have to be executed to retrieve data. The query itself is a string and thus cannot be checked for syntactical and type correctness. Thus the translation process should not only generate the backend code for writing the query as SQL string, it should also generate code that would ensure type safety of the generate query.

5.2.3 SQL backend design

We identified the need for type checking of the QuEL queries when converting to SQL, which introduces the open-source object relational mapping tool SQL2JAVA [14]. The tool is used to generate classes that will ensure type checking for the queries, and the query translation mechanism generates code which when complied and run will generate SQL query and execute the same on the concerned database and wraps the results appropriately in so that it could be iterated upon conveniently in the user code.

SQL2JAVA

The tool is freely available for download and comes with the executable for the open source HSQLDB [15] database. The tool can be configured to run `ant` [16] scripts that generate Java classes that map to tables in the underlying relational database schema. The database schema file should be provided to the tool for it to generate these classes, called Beans. The beans themselves provide additional functionality like persistence and transactional capabilities, the capability to execute simple queries such that from a programmer's perspective the beans can be treated as in-memory representation of the real database tables. However for the purpose of this thesis we shall not use the latter mentioned capabilities for limitations explained in the 5.2.3 section.

We assume for the purpose of doing QuEL translation into SQL using the designed backend,

the beans for mapping the tables into Java classes are generated before the preprocessing step and made available to the translation engine. Only the attributes of the beans will be used as they map to the columns of the table and in that regard any other relational–mapping tool could also be used to pregenerate these beans. Another alternative to using a tool could be to hand–write these beans by examining the database schema.

The design for the backend involves the following `Queryable<X> and Evaluate<X>` interface that serve as the contract for translation to SQL.
The `Evaluate<X>` interface is used to hide the body of method that chains the evaluation process once its trigerred.

```
public interface Evaluate<X> {

    /**
  * The method to that computes the resulting intermediate query
  * and determines if the query is complete and executes the same
  * on the database.
  *
  * @return An Iterable result of the query
  */
  public Iterable<X> eval() throws SQLException;
}
```

The `Queryable<X>` interface declares the publicly available extension methods of the backend. The translation mechanism calls on these methods to build the Java code equivalent of a QuEL query.

```
/**
* Takes in an <code>Expression</code> of the <code>SqlQuery</code>
```

```
* type
* and maintains the logic to convert this Expression into
* the SQL string appropriate time comes, i.e.  when the call to
* evaluate the result occurs.
* @author Anurag Naidu
*
* @param <X>
*/
public interface Queryable<X> extends Iterable<X> {
   /**
   * The select contains the columns to project and triggers the
   * evalution process and this is seen as logical termination of
   * complete QuEL query.
   *
   * @param select
   * @return An object of itself so that its a closed operation
   */
   public Queryable<X> select(Expression<SqlQuery> select);
   /**
   * The where contains the labmda condition that needs to
   * be checked as a query result filter.
   *
   * @param where
   * @return
   */
   public Queryable<X> where(Expression<SqlQuery> where);
   /**
   * The where contains the sources that are introduced into a
   * QuEL query, namely the database table names.
   *
   * @param from
   * @return
   */
   public Queryable<X> from(Expression<SqlQuery> from);
}
```

The hook in the `QuelInterpreter` is a subclass that translates the QuEL query towards the SQL backend for which it constructs an Expression¡X¿ object of from the following contract

```
public interface Expression<T> {
   /**
   * Function to generate the query string.  The string generated
   * so far is passed as input and the result is the same type
   * the parts of the query computer from the extension method
   * that defines this Expression class
   *
   * @param Parameter to the function (same as return type)
   * @return
   */
   T f (T param);
}
```

We can now look at the earlier QuEL query on relational database and compare with the translated code. The simple query that brings together the Staff and Rooms tables and returns all staff member who occupy rooms with size greater than 200 sq.ft.

```
final edu.rit.cs.quel.StaffBean staff = new
      edu.rit.cs.quel.StaffBean(Staff);

final edu.rit.cs.quel.RoomBean rooms = new
      edu.rit.cs.quel.RoomBean(Rooms);
@@
result = from edu.rit.cs.quel.StaffBean s in staff
      join edu.rit.cs.quel.RoomBean r in rooms on
         s.phone matches r.phone
      where r.size > 2000
      select new s.first, r.size
@@
```

Figure 18: QuEL query for database access

The query after translation is mapped to the `edu.rit.cs.sql.backend.Query` backend is converted to Java code below.

```
Queryable<ANON1548173586> result1 = new
edu.rit.cs.sql.backend.Query<ANON1548173586>(
   ANON1548173586.class, this, null).from(
      new Expression<SqlQuery>() {
      public SqlQuery f(final SqlQuery current) {
         current.fromClause.append("staff s, ");
         return current;
      }
   }).from(new Expression<SqlQuery>() {
      public SqlQuery f(final SqlQuery current) {
         current.fromClause.append("rooms r, ");
         return current;
      }
   }).where(new Expression<SqlQuery>() {
      public SqlQuery f(final SqlQuery current) {
         current.partialQueryClause.append("where ");
         current.partialQueryClause.append("r.size");
         current.partialQueryClause.append(">");
         current.partialQueryClause.append("200");
         return current;
      }
   }).select(new Expression<SqlQuery>() {
      public SqlQuery f(final SqlQuery current) {
         current.selectClause.append("select ");
         current.selectClause.append("s.name, r.size,");
         return current;
      }
   });
```

Figure 19: Java translation for QuEL query for database

Notice in the converted query how the method definition for the `Expression<T>` interface has the code to write out its part of the SQL query string. When invoked they contribute their part of the query to the `SqlQuery` object.

The `eval()` method returned from each of the extension methods `select, from, where,`

`join` etc of `Evaluate<X>` interface triggers the `T f (T param)` method passed to it as the first step which cases the chain to first construct the correct SQL string equivalent of the QuEL query. The methods `select or groupBy` carry special meaning as they execute the query on the database using the internal connection object. in Figure 19 above the code generated for type–safety checks is omitted from output now for brevity, however the Java code generated also has the type-safety check code that does not ever get executed but is merely there as a proof, that if the generated Java file compiles then the query is type–safe.

6 Comparison and Results

This chapter considers a few QuEL queries over XML and relational databases and compare the results from the respective backend against the results from a native language translation of the same query. As discussed in the backend design for relational database, the QuEL query is converted to the SQL equivalent and one of the measure of success for the backend is the comparison between the backend generated query and original QuEL query hand–written as SQL.

6.1 QuEL query translation for XML

Assume that we have the same *Staff* and *Rooms* XML documents from Figure 3 and we want to merge the two documents on the matching phone numbers and create a new document using the *name* and *room* values after grouping them based on *room* numbers and alphabetically reverse sorting the names within each group. The QuEL query to do the same would like the following:

```
@@
result = from s in elements(staff)
      join r in elements(rooms) on
      s.getElementsByTagName("phone").item(0).getTextContent() matches
      r.getElementsByTagName("phone").item(0).getTextContent() into sr
         select new { s.getElementsByTagName("first").item(0),
            @room = r.getAttribute("room")
         group by r.getAttribute("room")
         order by desc s.getTextContent()
@@
```

Figure 20: Quel query for XML

The translated query returns the following result which matches with the same query written using Java APIs for XML processing involving many more lines of code than the QuEL query.

```
Group groupedBy='3012'
   first room='3012'
      'Tina'
Group groupedBy='3599'
   first room=3599
      'James'
Group groupedBy='3005'
   first room='3005'
      'Eileen'
   first room='3005'
      'Jason'
Group groupedBy='3022'
   first room='3022'
      'Christina'
   first room='3022'
      'Liane'
```

Figure 21: XML query result

We will skip the translated Java code using XML backend from this text due to the shere size of the code, however it is interesting to note how this concise QuEL query to generate this results expands almost ten times its size when converted to pure Java using the XML backend and when compared with code written using only the available Java APIs for XML is roughly twenty times the length of the actual QuEL query. This only goes to show much syntactic sugar QuEL along with the XML backend can provide to a Java developer

interacting with XML documents.

6.2 XQuery for Java

There are a some XQuery [7] implementations that are avaliable for use in Java as API calls. One of them is OJXQI [17] proposed by Oracle [4] designed on the lines of JDBC [18] with similar APIs. OJXQI could be used to query XML documents by preparing a `PreparedXQuery` object and constructing an XQuery queries with it.

OJXQI being similar to JDBC in Java, has some advantages and similar disadvantages discussed in chapter 2 on XML which this thesis attempts to alleviate. However OJXQI also allows support for SQL queries on documents through a `sqlquery` function. These features bring it close the syntax of QuEL, however using OJXQI still involves writing boilerplate setup code just like in JDBC.

6.3 QuEL query translation for SQL

Consider a QuEL query to retrieve data from a relational database, in this case HSQLDB [15], that has the tables that we will query. Assume that we have the *Staff* and *Rooms* tables from Figure 8 in our database and we join on the *phone* column and then filter out rooms on the condition that they are greater than 2020 sq.ft. and numbered above 2000 with the results grouped by romms with same size. A SQL query to do the same would look like,

```
select s.*, r.size
from rooms r
inner join staff s on s.phone = r.phone
where r.number > 2000
```

```
and r.size > 2020
order by r.size
```

The same query written in QuEL would then look like,

```
...@@
   result = from sql2java.linq.database.RoomsBean r in rooms
         join sql2java.linq.database.StaffBean s in staff
            on r.phone matches s.phone
         where r.size > 2020
         where r.number > 2000
         orderby sr.size
         select new  sr.name, sr.number, r.size
@@ ...
```

The SQL backend converts this query into the following SQL,

```
select s.name, s.phone, r.size from rooms r
join staff s on s.phone = r.phone
where r.number > 2000 and r.size > 2020
order by r.size
```

The result of executing the QuEL query is the same as executing our first SQL query,

name=Jason Morrison phone=4010 size=2030

which confirms that the SQL backend does the appropriate translation.

6.3.1 GroupJoin issue

One of the limitations of the QuEL translation mechanism for SQL is with the `into` clause with the `join` and `groupby` clauses. The following QuEL query converts to the appropriate SQL however wrapping the `ResultSet` into the correct return type class fails.

```
...@@
   result = from sql2java.linq.database.RoomsBean r in rooms
         join sql2java.linq.database.StaffBean s in staff
            on r.phone matches s.phone into sr
         select new sr, r.number
@@ ...
```

The translated SQL appears like the following

```
select sr.*, r.number
from (
      select s.name, r.*
      from rooms r
      inner join staff s on s.phone = r.phone
   ) sr,
rooms r
```

This query executes on the database however the because the `select` clause specifies the variable `sr` as one of the return types the QuEL mapper [1] creates the return type class for the whole query with an instance of `StaffBean` wrapped in a `IterableImpl`. This is however insufficient for the purpose of SQL since use of `sr` in the select clause is equivalent to selecting all the columns of the table *Staff* and *Rooms*. This makes the return type object inconsistent with all the column data returned from the executed SQL.

7 Conclusion

Reconsidering the problem areas that we identified earlier with writing a query over XML using XPath and DOM APIs or over a relational database using SQL in Java and comparing how the QuEL way of querying on XML and SQL backend solves these problems.

- Complex code logic needed for fairly simple query patterns

  A query for performing the same operation is simpler, cleaner and more intuitive.

- Proficiency in many technologies like XPath, XSLT, JDK XPath APIs is needed.

  The programmer only needs to learn QuEL for the most part for writing queries over any of the supported data sources.

- Repetitive boilerplate code to supplement the actual query code.

  No boilerplate code is written by the programmer. The backend for the respective data source technology will auto generate all boilerplate code and keep the query logic clean.

- The query itself remains type unchecked and so potentially unsafe.

  The query is not checked for type safety by the backend. Since the backend generates pure Java code that is compiles, thus the queries are themselves are assured of being type checked otherwise the code generated by the backend would not have compiled.

- Lack of consistent query language for all data–sources.

  QuEL grammar does not change significantly with the data–source it operates on and thus querying any source using QuEL is always the same consistent experience.

7.1 Timing

The results of the QuEL queries that were executed were only compared for their correctness and not for the query execution time. The comparison also took into account with sematic similarity of the quEL query after it went through the translation using the respective back–end. However, since reflection is extensively used in both the query back–end designed in this thesis, some performance cost has to accounted for the same. Also both, the XML and relational database, back–end leverage the DOM and JDBC APIs respectively to compute the result internally, thus the performance of the QuEL queries will never improve the performance of writing these queries using these APIs directly. Considering these two factors, the performance of QuEL queries is always slightly, if not significantly, lower than using the DOM and JDBC APIs directly. This could just well be a small price to pay for the syntactic sugar that a programmer getfrom using QuEL for XML and relational databases.

The most significant contribution of this work would be in overall all improvement in application development time because of the type–checking that it offers and that because now significantly lesser, 50% on an average, lines of code needs to be written for querying.

8 Limitations

The backend designed for XML and relational databases extends the QuEL language to be used for writing queries for those data–sources within the Java programming language without the additional burden of writing setup code required for using the APIs in JDK for accessing these sources. However there are some areas in the translation that do not conform to the rules of the native query language for that data–source.

8.1 Into

One such problem highlighted earlier was that of the `into` clause, when translated to SQL creates ambiguity for the names of the columns that are projected. The SQL can be made to select all the columns using `table.*` syntax, however the backend system does not have the columns names to in order for it to create the result object correctly. The `Into` clause when used with `join` and `groupby` is plagued by this issue, but not the `select` clause if the projected column names are clearly listed out in the clause.

Therefore the following query syntax does not work for relational database queries,

```
... join sql2java.linq.database.StaffBean s in staff
     on r.phone matches s.phone into sr
```

and

```
... group r by r.size into sr
```

In order to get around this problem the programmer should avoid using the `into` alias in such places. Although this does not provide the flexibility of temporarily storing the

results of a `join` and `groupby` clause in an alias, it does not limit the ability of the query itself to retrieve the correct data.

8.2 Timing

As discussed in the in the conclusions, no query execution performance improvements can be gained from using QuEL. QuEL was designed with the intent that developer writing queries for a data–source would have to write less Java code, and thus for scenarios where the data–sources queried are enormous in size and if the performance in these scenarios is extremely critical to the application, QuEL would not serve the desired purpose of achieving faster query response time.

9 Resources

- **Java 6:** The preprocessor will be written using Java 6 (JDK 1.6) release. The "@@" symbol is reserved as the preprocessor directive for the new compiler and unless this is ever used as a special token the preprocessor will support all future versions is Java.

- **QuEL:** The new query grammar definition and translation mechanism developed in the QuEL [1] thesis by Aaron R. Robinson.

- **Query backend:** The query backend for Java collections will be reused for converting queries on relational databases. This backend was written by Dr. Axel Schreiner for the purpose of implementing a LINQ [12] like language for Java collections.

- **SQL2JAVA:** An open-source project for object-relational mapping a relational database to classes in the OO programming world.

- **HSQLDB:** An open-source database engine now part of OpenOffice 2.0 release. This is used as the development/test database for creating and running Linq queries on database entities.

References

[1] Aaron R. Robinson. Integrating a universal querying mechanism in java. http://www.rit.edu/ arr9595/, March 2007.

[2] Extensible markup language (xml). http://www.w3.org/XML/, 1996.

[3] Don Box and Anders Hejlsberg. The LINQ project .NET Language Integrated Query. http://www.microsoft.com/downloads/details.aspx?familyid=1e902c21-340c-4d13-9f04-70eb5e3dceea, May 2006.

[4] Oracle 9i sql reference release 2. http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96540/toc.h 2005.

[5] W3c xml path language (xpath) (v1.0). http://www.w3.org/TR/xpath, 1999.

[6] W3C. Document object model (dom). http://www.w3.org/DOM/, 2001.

[7] Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery/, 2005.

[8] Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt, 1999.

[9] E.F Codd. The relational model for database management. Addison Wesley Publishing Company. ISBN 0-201-14192-2, 1990.

[10] Microsoft Corporation. Language integrated query. http://en.wikipedia.org/wiki/Language_Integrated_Query, 2007.

[11] Extension method. http://en.wikipedia.org/wiki/Extension_method, 2007.

[12] Dr. Axel T. Schreiner. Linq – why we have to teach functional programming. http://www.cs.rit.edu/ ats/talks/linq/linq/Query.java, November 2006.

[13] Dr. Michael Kay. Schema-aware queries and stylesheets. http://www.stylusstudio.com/schema_aware.html.

[14] Sql2java a free, open source object-relational mapping tool. http://sql2java.sourceforge.net/, 2005.

[15] Hsqldb - 100% java database. http://hsqldb.org/, 2007.

[16] Apache Corporation. The apache ant project. http://ant.apache.org/.

[17] Oracle Corporation. Ojxqi - the oracle java xquery api. http://www.oracle.com/technology/sample_code/tech/xml/xmldb/jxqi.html.

[18] The java database connectivity (jdbc). http://java.sun.com/javase/technologies/database/, 1999.