Performance of an XML DBMS as a Model Repository in Model-Driven Software Development

Submitted by
Angelina Velinska
an.velinska@gmail.com
Information and Media Technologies
Matriculation Number
20729536

Supervised by
Prof. Dr. Ralf Möller
STS - TUHH

M.Sc. Miguel Garcia
STS - TUHH

Hamburg, Germany
2009-02-17

Abstract

Model repositories are Object Oriented DBMSs that are used to persist Ecore EMF Objects. EMF Objects can be queried and retrieved by means of the language LINQ, these queries are in turn translated into the native query language of the database engine of choice e.g., LINQ to XQuery translation. For querying and managing EMF Objects in DBMS, one possibility is to use a language that can be processed on the database without a need of further translation. Such a query language is XQuery, which can be processed by the XML database management system eXist db. The performance of XQuery queries on eXist database is evaluated, used as a model repository for large EMF models.

Declaration

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

Hamburg, February 17, 2009
Angelina Velinska

Acknowledgements

I would like to thank sincerely Prof. Dr. Ralf Möller for giving me the opportunity to do my project work in the Institute for Software Systems (STS) at the Hamburg University of Technology (TUHH), Germany.

I am also very thankful to M.Sc. Miguel Garcia, for supervising this project work, for his patience and valuable guidelines that proved critical for the successful completion of this work.

Hamburg, February 17, 2009
Angelina Velinska

Contents

Chapter 1

Introduction

Summary.

This chapter introduces the concepts of Ecore EMF as a modeling and code-generating framework, and makes an overview of the basic types of XML DBMSs. It presents shortly the query languages LINQ and XQuery, and concludes by summarizing the objective of this project work, and the contents of the project work, divided by chapters.

1.1 Eclipse EMF

Model-Driven Architecture concepts [OV08] have been around for quite a while now and currently many software platforms, frameworks, tools, and applications work with models and model repositories. Various tools and frameworks provide a wide range of services for operation with models: transformation languages (both model-to-model and model-to-text), model querying and validation facilities, different persistence solutions, code generation, etc.

One of the most popular modeling frameworks currently is Eclipse Modeling Framework (EMF)[1]. EMF has gone its way of development and relies on its own metamodel ECore.It has quickly become the de facto standard of model handling in Eclipse-based tools, and is also becoming popular outside Eclipse in standalone applications. It is widely used as a tool for the implementation of a structured model. EMF has a rich set of services available and thus is an appealing environment for model handling. Existing applications can benefit from allowing their models to be transferred to EMF and back. For example, such interoperability could add missing features to existing model environments when needed, enabling XMI and XML model serialization (serialization mechanisms provided by EMF), validation of the model against a defined rule set, code generation functionality or model-to-model transformations, etc. Applications such as DBMSs can also benefit from integration with EMF using the additional services from EMF-based tools.

[1] `http://www.eclipse.org/modeling/emf/`

1.2 XML Database Managements Systems

1.2.1 Types of XML DBMSs

The eXtensible Markup Language (XML) has gained broad popularity today and is widely used. It is supported by all vendors and is universal. It is used in a variety of ways, among which:

1. Publishing of data stored in an existing database as XML.
2. Transfering of data from applications to databases in the form of an XML document
3. Modeling of semi-structured data, as XML is extensible.

XML documents are used to exchange data between a database and an application or another database. The process of extracting data from a database and constructing an XML document (or XML fragments) is known as *publishing* or *composition*. The reverse process (extracting data from an XML document (or XML fragment) and storing it in the database) is known as *shredding* or *decomposition*.

XML documents are stored in a database. In **XML-enabled databases**, there is no XML "visible" inside the database. That is, the XML document is completely external to the database. It is constructed from data already in the database or is used as a source of new data to store in the database. Also, the database schema matches the XML schema. That is, a different XML schema is needed for each database schema.

On the other hand, if the document itself is stored in the database, it can be stored in a set of tables designed especially to store XML documents. The database generated document IDs. They do not necessarily contain data extracted from the stored documents themselves. In this case, the XML is "visible" inside the database. That is, the database contains information such as element type and attribute names. Furthermore, a single database schema can be used to store all XML documents. In other words, the database schema models XML documents, not the data in those documents. Databases that use XML in this fashion are known as **native XML databases**.

A more technically correct view of the difference between XML-enabled databases and native XML databases entails that XML adds a new data model to the world of databases. That is, in addition to the existing hierarchical, relational, object-oriented, multi-valued, and other data models, XML adds its own model. The XML data model is *an ordered tree with typed, labeled branch nodes and data stored in unlabeled leaf nodes.*

In an *XML-enabled database*, existing data can be used to create XML documents, a process known as *publishing*. Similarly, data from an XML document can be stored in the database, a process known as *shredding*. In an *XML-enabled database*, no XML is "visible" inside the database and the database schema must be mapped to an XML schema. XML-enabled databases are used when XML is used as a data exchange format and requires no changes to existing applications.

In a *native XML database*, a set of generic structures is used to store any and all XML documents. Because of this, XML is "visible" inside the database. No schema mapping is necessary since native XML databases use the XML data model directly.

Native XML databases support an XML query language and can be built from scratch or on top of an existing database. They are best used for storing documents (such as user's manuals, static Web pages, and marketing brochures) and semi-structured data.

1.2.2 Benchmarking of XML databases

XML-Applicatons process increasingly big to huge collections of XML documents. Therefore, it is a challenge to the efficiency of persisting data and posting queries on the database. Benchmarks are an objective mean to measure the performance and efficiency of the work of a system. As the various XML-applications are conforming to specific requirements, at the search of a suitable benchmark one should have in mind his/ her own problem and choose the suitable Benchmark. There are minimum four criteria[Ben93] that the Benchmark should support:

1. It should be **relevant**: the Benchmark should define the optimal performance of the system and the price-performance ratio during execution of typical operations of the application.

2. It should be **portable**: the implementation should be simple and should be capable to work under multiple systems and architectures.

3. It should be **scalable**: The benchmark should be able to work with small systems containing little data, as well as with big system installations with large amounts of data.

4. It should be **simple**: The results should be understandabe and simple to interpret.

Examples of Benchmarks that have proven good for the performance analysis of XML-processing of data, are Xmark[Ams], XOO7[Sin], XMach-1[Lei].

1.3 XQuery and LINQ

Developing applications that work with database systems has become common in software development, and hence the developer has to deal with two different worlds, object-oriented domain models and physical database systems. Different query languages exist to handle these different models. As LINQ can be used to query the logical entity model, also XQuery can be used to query XML database systems. A lot of efforts are being made to abstract the physical data model as much as possible, so that the developer can concentrate only on object domain model. At runtime, queries expressed over the object model can be translated into the native query language (XQuery) of the target data store by a component contributed by the DBMS vendor. [Pri08]

Language Integrated Query (LINQ) is a Microsoft .NET Framework component that adds native data querying capabilities to .NET languages like C#, VB. LINQ defines a set of query operators that can be used to query, project and filter data in arrays, enumerable classes, XML, relational database, and third party data sources. As a Microsoft .NET Framework component, LINQ is widely used with .NET languages such as C# and Visual Basic. XQuery on the other hand is the standard query language for XML databases. It is a strongly typed functional language that supports both the transformation and querying of XML documents. It allows the extraction of data

through XPath expressions, the joining of several documents, and the construction of completely new documents. In this project, we will post translated LINQ to XQuery queries to eXist XML database in order to benchmark the performance results.

1.4 Objective

The main objectives of this project work are:

1. Generation of a large Ecore data model in Eclipse EMF and its export as an XML document. The exported data is imported then into an XML DBMS eXist, and the performance of XQuery queries, posted on the data is evaluated.
2. Translation of LINQ go XQuery queries to be posted on eXist-db data. This is concidered as an interim step aiming at the ultimate goal of a realization of a LINQ for Java provider.

1.5 Organization of the Report

This project work will proceed in the following way:

- In **Chapter** 1 we gave a short overview of Eclipse EMF framework, the different types of XML databases, benchmarking of XML DBMSs, LINQ and XQuery languages.
- In **Chapter** 2 we will present Eclipse EMF framework and will create and instantiate our data model, using EMF.
- In **Chapter** 3 will be made a LINQ to XQuery translation of the queries, which will be posted on our generated data, imported in an XML database.
- **Chapter** 4 will present an overview of the benchmarking on XML DBMS, and in particular on eXist db, using our custom data model.
- **Chapter** 5 will make an overview and will give some ideas for future works.

Chapter 2

EMF Used as a Model Generator

Summary. *This chapter briefs on the generation of a data model in Eclipse EMF, defined as an XML Schema. Following its instantiation, the model is exported as XML document to be used in the benchmarking of eXist db.*

Ed Merks, Lead of the Eclipse Modeling Framework project:
"Although you might not realize it, models drive most software e.g., dealing with data models in business applications. However, using source code as a representation for your models might not always be the best solution - to gain both a better overview of your software and be more efficient with respect to software development, you should think about using domain specific modeling tools that let you focus on the structure of your model."

2.1 Introduction

EMF[EMF08] is a powerful framework and code generation facility for building Java applications based on simple model definitions. Designed to make modeling practical and useful to the mainstream Java programmer, EMF unifies three important technologies: Java, XML, and UML. Models can be defined using a UML modeling tool or an XML Schema, or even by specifying simple annotations on Java interfaces. In this last case, the developer writes just a subset of abstract interfaces that describe the model, and the rest of the code is generated automatically and merged back in.

By relating modeling concepts to the simple Java representations of those concepts, EMF has successfully bridged the gap between modelers and Java programmers. It serves as an introduction to modeling for Java programmers and at the same time as a reinforcement of the modeler's theory that a great deal of coding can be automated, given an appropriate tool.

EMF provides a runtime framework that allows any modeled data to be easily validated, persisted, and edited in a user interface. Change notification and recording

are supported automatically. Metadata is available to enable generic processing of any data using a uniform, reflective API. With all of these features and more, EMF is the foundation for data sharing and fine-grained interoperability among tools and applications in Eclipse, in much the same way that Eclipse is itself a platform for integration at the component and UI level.

2.2 Motivation and Model Generation

EMF framework offers comparatively easy and practical modeling for Java projects. It also provides interoperability and data sharing accross multiple toos and platforms. Due to its flexibility in defining models, and its ability to serialize the generated models as XMI or XML, it was chosen for the purpose of model generation in our project.

The model is based on Microsoft's Programming LINQ [Lin08]. The model is defined and imported in EMF as XML Schema, in order to achieve a better control on the serialized output. Listing 2.1 shows an excerpt of the XML Schema used as the model definition.

When instances of an Ecore model are created and serialized, they are serialized by default as XMI 2.0[1]. We took advantage of the option to serialize intances of Ecore model as XML documents, which are better suited for import into an XML DBMS afterwards. A clear step-by-step introduction on generating models in EMF from XML Schema is given in [Gui].

Listing 2.1: XML Schema Decsribing the Model

```
1  <xsd:complexType name="Customer">
2    <xsd:sequence>
3      <xsd:element maxOccurs="unbounded" minOccurs="0" name="order"
4          type="cus:Order"/>
5    </xsd:sequence>
6    <xsd:attribute name="id" type="xsd:integer" use="required"  />
7    <xsd:attribute name="name" type="xsd:string" use="required" />
8    <xsd:attribute name="city" type="xsd:string" use="required" />
9    <xsd:attribute name="country" type="cus:Country"  use="optional"/>
10 </xsd:complexType>
11 <xsd:complexType name="customers">
12   <xsd:sequence>
13     <xsd:element name="customer" type="cus:Customer" minOccurs="0"
         maxOccurs="unbounded"/>
14   </xsd:sequence>
15 </xsd:complexType>
```

Once the model is imported in EMF, and the Model, Edit and Editor code are generated by the framework, we can make various adjustments and customizations, from changing the implementation of the generated model classes to extending and customizing the editors. The customizations and changes are implemented during regeneration of the Model, if the `@generated` *JavaDoc* tag is removed from the modified source sode, so that EMF's *jmerge* will not overwrite our custom method, attribute, or class.

[1] `http://www.omg.org/technology/documents/formal/xmi.htm`

2.3 Manipulating the Generated Model

After the model is defined and generated, EMF allows us to create and save an instance of the model, resulting in an output XML document. Listing 2.2 shows the creation of a Cus model instance and its population with data.

Listing 2.2: Serialization of the Model

```
1  //  Initialization of objects.
2          CusFactory factory = CusFactory.eINSTANCE;
3          Customers customers = factory.createCustomers();
4          Customer customer1 = factory.createCustomer();
5          Order order1 = factory.createOrder();
6          Product product1 = factory.createProduct();
7          customer1.setName("John Smith");
8          customer1.setCity("San Francisco");
9          customer1.setCountry(Country.USA);
10         order1.setId(new Integer(1));
11         product1.setName("Beer");
12         product1.setPrice(new BigDecimal(3.5));
13
14 //  Add the objects to the root element.
15         customers.getCustomer().add(customer1);
16         customer1.getOrder().add(order1);
17         order1.getProduct().add(product1);
18
19 //  Add the root element to the resource.
20         resource.getContents().add(customers);
21
22 //  Get the URI of the model file.
23         URI fileURI = URI.createFileURI(new File("Cus.xml").
               getAbsolutePath());
24
25 //  Create a resource to serialize the model, set the encoding.
26         XMLResource resource = new XMLResourceImpl(fileURI);
27         resource.setEncoding("UTF-8");
28
29 //  Save the contents of the resource to the file system.
30         resource.save(options);
```

In this example, we first obtain an instance of the CusFactory, and then use it to create all of the subobjects. Once created, they are all added to the root element `customers`. Because the target serialization format is XML, we use the `XMLResourceFactoryImpl`. After the generation of our data model, we created a data document containing 1000 `customer` records.

Once we have created and manipulated the EMF model, it is saved to the format of our choosing XML (see Listing 2.6).

Listing 2.3: The Serialized Model

```
<?xml version="1.0" encoding="UTF-8"?>
<cus:customers xmlns:cus="http://schemas.devleap.com/cus">
  <customer city="San Francisco" country="USA" id="1" name="John Smith">
    <order id="11" month="2" shipped="false">
      <product id="111" name="Beer">
        <price>3.50</price>
      </product>
      <product id="112" name="Juice">
        <price>2.30</price>
      </product>
    </order>
    <order id="12" month="3" shipped="true">
      <product id="121" name="Campari">
        <price>5.41</price>
      </product>
    </order>
  </customer>
  <customer city="Firenze" country="Italy" id="2" name="Paolo Franco">
    <order id="21" month="2" shipped="true">
      <product id="221" name="Pasta">
        <price>2.75</price>
      </product>
    </order>
  </customer>
</cus:customers>
```

2.4 Appendix

Listing 2.4: XML Schema of the Data Model

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema
    targetNamespace="http://schemas.devleap.com/cus"
    xmlns:cus="http://schemas.devleap.com/cus"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:simpleType name="Country">
        <xsd:restriction base="xsd:NCName">
          <xsd:enumeration value="Italy" />
          <xsd:enumeration value="USA" />
          <xsd:enumeration value="Germany" />
        </xsd:restriction>
     </xsd:simpleType>
     <xsd:complexType name="Product" >
       <xsd:choice>
                <xsd:element name="price" type="xsd:decimal"
            minOccurs="1" maxOccurs="1"/>
       </xsd:choice>
       <xsd:attribute name="id" type="xsd:integer" use="required" />
       <xsd:attribute name="name" type="xsd:string" use="optional"/>
     </xsd:complexType>
         <xsd:complexType name="Order">
           <xsd:sequence>
             <xsd:element maxOccurs="unbounded" minOccurs="1"
             name="product" type="cus:Product" />
           </xsd:sequence>
           <xsd:attribute name="id" type="xsd:integer" use="required" />
       <xsd:attribute name="shipped" type="xsd:boolean" use="required"/>
       <xsd:attribute name="month" type="xsd:int" use="optional" />
         </xsd:complexType>
         <xsd:complexType name="Customer">
       <xsd:sequence>
         <xsd:element maxOccurs="unbounded" minOccurs="0" name="order"
              type="cus:Order"/>
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:integer" use="required"  />
       <xsd:attribute name="name" type="xsd:string" use="required" />
       <xsd:attribute name="city" type="xsd:string" use="required" />
       <xsd:attribute name="country" type="cus:Country" use="optional"/>
         </xsd:complexType>
     <xsd:complexType name="customers">
       <xsd:sequence>
         <xsd:element name="customer" type="cus:Customer" minOccurs="0"
              maxOccurs="unbounded"/>
       </xsd:sequence>
     </xsd:complexType>
</xsd:schema>
```

Listing 2.5: Intialization of the Data Model

```
1  package com.devleap.schemas.cus;
2
3  public class Initialization {
4  public static void main(String[] args){
5
6          CusPackage cp = CusPackage.eINSTANCE;
7          printClasses(cp);
8          Initialization.init();
9  }
10
11 public static void init(){
12
13 //  Initialization of objects with 'real' data.
14         CusFactory factory = CusFactory.eINSTANCE;
15         Customers customers = factory.createCustomers();
16         Customer customer1 = factory.createCustomer();
17         Order order11 = factory.createOrder();
18         Order order12 = factory.createOrder();
19         Order order13 = factory.createOrder();
20         Product product111 = factory.createProduct();
21         Product product112 = factory.createProduct();
22         Product product113 = factory.createProduct();
23         Product product114 = factory.createProduct();
24         Product product115 = factory.createProduct();
25         Product product116 = factory.createProduct();
26         Product product117 = factory.createProduct();
27         Product product118 = factory.createProduct();
28         Product product121 = factory.createProduct();
29         Product product122 = factory.createProduct();
30         Product product123 = factory.createProduct();
31         Product product124 = factory.createProduct();
32         Product product125 = factory.createProduct();
33         Product product131 = factory.createProduct();
34         Product product132 = factory.createProduct();
35         Product product133 = factory.createProduct();
36
37         customer1.setId(1);
38         customer1.setName("John Smith");
39         customer1.setCity("San Francisco");
40         customer1.setCountry(Country.USA);
41         order11.setId(new Integer(11));
42         order11.setShipped(false);
43         order11.setMonth(2);
44         order12.setId(new Integer(12));
45         order12.setShipped(true);
46         order12.setMonth(3);
47         order13.setId(new Integer(13));
48         order13.setShipped(true);
49         order13.setMonth(4);
50         product111.setId(111);
51         product111.setName("Beer");
52         product111
53         .setPrice(new BigDecimal(3.5).setScale(2, BigDecimal.ROUND_UP));
54         product112.setId(112);
55         product112.setName("Juice");
56         product112
57         .setPrice(new BigDecimal(2.3).setScale(2, BigDecimal.ROUND_UP));
58         product113.setId(113);
59         product113.setName("Cream");
60         product113
61         .setPrice(new BigDecimal(0.8).setScale(2, BigDecimal.ROUND_UP));
62         product114.setId(114);
63         product114.setName("Pasta");
64         product114
```

```
65          .setPrice(new BigDecimal(2).setScale(2, BigDecimal.ROUND_UP));
66          product115.setId(115);
67          product115.setName("Pop-corn");
68          product115
69          .setPrice(new BigDecimal(1).setScale(2, BigDecimal.ROUND_UP));
70          product116.setId(116);
71          product116.setName("Sausage");
72          product116
73          .setPrice(new BigDecimal(4.3).setScale(2, BigDecimal.ROUND_UP));
74          product117.setId(117);
75          product117.setName("White wine");
76          product117
77          .setPrice(new BigDecimal(3.9).setScale(2, BigDecimal.ROUND_UP));
78          product118.setId(118);
79          product118.setName("Cogniak");
80          product118
81          .setPrice(new BigDecimal(7.9).setScale(2, BigDecimal.ROUND_UP));
82          product121.setId(121);
83          product121.setName("Campari");
84          product121
85          .setPrice(new BigDecimal(5.4).setScale(2, BigDecimal.ROUND_UP));
86          product122.setId(122);
87          product122.setName("Coca Cola");
88          product122
89          .setPrice(new BigDecimal(2.5).setScale(2, BigDecimal.ROUND_UP));
90          product123.setId(123);
91          product123.setName("Butterpastry");
92          product123
93          .setPrice(new BigDecimal(1.9).setScale(2, BigDecimal.ROUND_UP));
94          product124.setId(124);
95          product124.setName("Vegetable Mix");
96          product124
97          .setPrice(new BigDecimal(3.4).setScale(2, BigDecimal.ROUND_UP));
98          product125.setId(125);
99          product125.setName("Vodka");
100         product125
101         .setPrice(new BigDecimal(4.8).setScale(2, BigDecimal.ROUND_UP));
102         product131.setId(131);
103         product131.setName("Chips");
104         product131
105         .setPrice(new BigDecimal(1.8).setScale(2, BigDecimal.ROUND_UP));
106         product132.setId(132);
107         product132.setName("Muffin");
108         product132
109         .setPrice(new BigDecimal(1.3).setScale(2, BigDecimal.ROUND_UP));
110         product133.setId(133);
111         product133.setName("Chicken Breasts");
112         product133
113         .setPrice(new BigDecimal(4.8).setScale(2, BigDecimal.ROUND_UP));
114         Customer customer2 = factory.createCustomer();
115         Order order21 = factory.createOrder();
116         Product product221 = factory.createProduct();
117         customer2.setId(2);
118         customer2.setName("Paolo Franco");
119         customer2.setCity("Firenze");
120         customer2.setCountry(Country.ITALY);
121         order21.setId(new Integer(21));
122         order21.setMonth(2);
123         order21.setShipped(true);
124         product221.setId(221);
125         product221.setName("Pasta");
126         product221
127         .setPrice(new BigDecimal(2.75)
128         .setScale(2, BigDecimal.ROUND_UP));
129
```

```
130         // Add the objects to the root element.
131         customers.getCustomer().add(customer1);
132         order11.getProduct().add(product111);
133         order11.getProduct().add(product112);
134         order11.getProduct().add(product113);
135         order11.getProduct().add(product114);
136         order11.getProduct().add(product115);
137         order11.getProduct().add(product116);
138         order11.getProduct().add(product117);
139         order11.getProduct().add(product118);
140         customer1.getOrder().add(order11);
141         order12.getProduct().add(product121);
142         order12.getProduct().add(product122);
143         order12.getProduct().add(product123);
144         order12.getProduct().add(product124);
145         order12.getProduct().add(product125);
146         customer1.getOrder().add(order12);
147         order13.getProduct().add(product131);
148         order13.getProduct().add(product132);
149         order13.getProduct().add(product133);
150         customer1.getOrder().add(order13);
151         order21.getProduct().add(product221);
152         customer2.getOrder().add(order21);
153         customers.getCustomer().add(customer2);
154
155 //  Get the URI of the model file.
156         URI fileURI = URI.createFileURI(new File("Cus.xml").
                getAbsolutePath());
157
158 //  Create a resource to serialize the model, set the encoding.
159         XMLResource resource = new XMLResourceImpl(fileURI);
160         resource.setEncoding("UTF-8");
161
162 //  Specify save options to serialize the output.
163         HashMap options = new HashMap();
164         options
165         .put(XMLResource.OPTION_KEEP_DEFAULT_CONTENT, Boolean.TRUE);
166         options
167         .put(XMLResource.OPTION_EXTENDED_META_DATA, Boolean.TRUE);
168
169 //   Initialization of objects with 'bogus' data
170 //       for creation of a large data collection.
171         for(int i=3; i<1001; i++){
172           Customer customer = factory.createCustomer();
173           customer.setName("customerName" + i);
174           customer.setCity("city" + i);
175           if (i % 2 == 0){
176            customer.setCountry(Country.GERMANY);}
177           else if (i%3 == 0)
178           {customer.setCountry(Country.ITALY);}
179           else {customer.setCountry(Country.USA);}
180            customer.setId(i);
181
182           Order order = factory.createOrder();
183           order.setId(10 + i);
184           if (i < 13){
185                 order.setMonth(i);
186           }else if (i % 13 == 0){
187                 order.setMonth(1);
188           }else {order.setMonth(i % 13);
189           }
190           if (i%3==0)
191           {order.setShipped(false);
192           }else {order.setShipped(true);
193           }
```

```
194
195             Product product = factory.createProduct();
196             product.setId(i);
197             product.setName("name"+i);
198             product.setPrice
199                     (new BigDecimal(i).setScale(2, BigDecimal.ROUND_UP));
200             order.getProduct().add(product);
201             customer.getOrder().add(order);
202             customers.getCustomer().add(customer);
203             resource.getContents().add(customers);
204 //  Save the contents of the resource to the file system.
205 //  Each iteration saves 1 customer, 1 order, 1 product to the data file
        .
206                     try
207                     {
208                             resource.save(options);
209                     }
210                             catch (IOException e) {}
211                     }
212 //  Finish.
213                             System.out.println(fileURI);
214 }
215
216 // Iterates on the model and prints out the model EClasses
217 // (XML Elements) with their EReferences (XML attributes).
218 public static void printClasses(EPackage ePackage)
219 {
220   for (Iterator iter =
221        ePackage.getEClassifiers().iterator(); iter.hasNext(); ) {
222     EClassifier classifier = (EClassifier)iter.next();
223     System.out.println(classifier.getName());
224     System.out.print("  ");
225
226     if (classifier instanceof EClass) {
227       EClass eClass = (EClass)classifier;
228       for (Iterator ai =
229            eClass.getEAttributes().iterator(); ai.hasNext(); ) {
230         EAttribute attribute = (EAttribute)ai.next();
231         System.out.print(attribute.getName() + " ");
232       }
233       for (Iterator ri =
234            eClass.getEReferences().iterator(); ri.hasNext(); ) {
235         EReference reference = (EReference)ri.next();
236         System.out.print(reference.getName() + " ");
237       }
238     }
239     else if (classifier instanceof EEnum) {
240       EEnum eEnum = (EEnum)classifier;
241       for (Iterator ei =
242            eEnum.getELiterals().iterator(); ei.hasNext(); ) {
243         EEnumLiteral literal = (EEnumLiteral)ei.next();
244         System.out.print(literal.getName() + " ");
245       }
246     }
247     else if (classifier instanceof EDataType) {
248       EDataType eDataType = (EDataType)classifier;
249       System.out.print(eDataType.getInstanceClassName() + " ");
250     }
251   }
252  }
253 }
```

Listing 2.6: The Serialized Model

```
<?xml version="1.0" encoding="UTF-8"?>
<cus:customers xmlns:cus="http://schemas.devleap.com/cus">
  <customer city="San Francisco" country="USA" id="1" name="John Smith">
    <order id="11" month="2" shipped="false">
      <product id="111" name="Beer">
        <price>3.50</price>
      </product>
      <product id="112" name="Juice">
        <price>2.30</price>
      </product>
      <product id="113" name="Cream">
        <price>0.81</price>
      </product>
      <product id="114" name="Pasta">
        <price>2.00</price>
      </product>
      <product id="115" name="Pop-corn">
        <price>1.00</price>
      </product>
      <product id="116" name="Sausage">
        <price>4.30</price>
      </product>
      <product id="117" name="White wine">
        <price>3.90</price>
      </product>
      <product id="118" name="Cogniak">
        <price>7.91</price>
      </product>
    </order>
    <order id="12" month="3" shipped="true">
      <product id="121" name="Campari">
        <price>5.41</price>
      </product>
      <product id="122" name="Coca Cola">
        <price>2.50</price>
      </product>
      <product id="123" name="Butterpastry">
        <price>1.90</price>
      </product>
      <product id="124" name="Vegetable Mix">
        <price>3.40</price>
      </product>
      <product id="125" name="Vodka">
        <price>4.80</price>
      </product>
    </order>
    <order id="13" month="4" shipped="true">
      <product id="131" name="Chips">
        <price>1.81</price>
      </product>
      <product id="132" name="Muffin">
        <price>1.31</price>
      </product>
      <product id="133" name="Chicken Breasts">
        <price>4.80</price>
      </product>
    </order>
  </customer>
  <customer city="Firenze" country="Italy" id="2" name="Paolo Franco">
    <order id="21" month="2" shipped="true">
      <product id="221" name="Pasta">
        <price>2.75</price>
      </product>
    </order>
```

```
65    </customer>
66    <customer city="city3" country="Italy" id="3" name="customerName3">
67      <order id="13" month="3" shipped="false">
68        <product id="3" name="name3">
69          <price>3.00</price>
70        </product>
71      </order>
72    </customer>
73    <customer city="city4" country="Germany" id="4" name="customerName4">
74      <order id="14" month="4" shipped="true">
75        <product id="4" name="name4">
76          <price>4.00</price>
77        </product>
78      </order>
79    </customer>
80    <customer city="city5" country="USA" id="5" name="customerName5">
81      <order id="15" month="5" shipped="true">
82        <product id="5" name="name5">
83          <price>5.00</price>
84        </product>
85      </order>
86    </customer>
87
88    <customer>... records from 6 to 999 ...</customer>
89
90    <customer city="city1000" country="Germany" id="1000"
91    name="customerName1000">
92      <order id="1010" month="12" shipped="true">
93        <product id="1000" name="name1000">
94          <price>1000.00</price>
95        </product>
96      </order>
97    </customer>
98  </cus:customers>
```

Chapter 3

LINQ to XQuery Translation

Summary. *After we presented Eclipse EMF framework as a model generator, and instantiated our data model, in this chapter we will create the queries to be posted on the data, imported in an eXist database. The Queries are based on LINQ[Lin08] and are a LINQ-to-XQuery translation.*

3.1 Introduction and Related Works

This project is intended as an interim step to the realisation of a 'LINQ for Java' provider. Below are given 3 of the implementation of LINQ for Java that exist currently:

1. Quaere[NM08] - it is an internal DSL that adds a querying syntax similar to SQL to Java. The Quaere project allows users to use an internal DSL to filter, enumerate and create projections over a number of collections and other queryable resources using a common, expressive syntax.

2. JaQue[Tri08] or "JAva integrated QUEry library" - it provides an infrastructure for Microsoft LINQ like capabilities on Java platform. Using Java bytecode manipulation techniques, JaQue builds expression trees, which can be translated later to native query language of the database. JaQue library is designed to take advantage of the new Java 7 syntax - *closures*[1].

3. QueryDSL[Wes09] - it is a framework which enables the construction of statically typed SQL-like queries, constructed via a fluent DSL/API like Querydsl. The framework currently supports:

   - JPA / Hibernate backends
   - JDBC / SQL backends
   - Java Bean based collections and lists

LINQ aims at bridging the Object-relational impedance mismatch[2]. The advantages for implementing LINQ for the Java world among others, are: type-safety, expression tree composition, query capabilities added to Java language. LINQ is a pluggable

[1] `http://www.javac.info/`
[2] `http://www.agiledata.org/essays/impedanceMismatch.html`

technology, as it is based on so called *standard query operators*[3], which are not dependant on a certain data source. As long as these operators can be exposed to the data source of choice, it can be queried. LINQ is designed to work on any data and container type. Based on the its growing popularity, there already exist quite a few implementations of LINQ providers[4].

3.2 LINQ to XQuery Translation

3.2.1 `Where` Query

Listing 3.7 gives an example of a query that selects all customers from the database originally coming from Italy.

```
LINQ:
var expr =
        customers
        .Where((c, index) => (c.Country == Countries.Italy))
        .OrderByDescending(c => c.Name)
        .Select(c => new {c.Name, c.City});

XQuery:
declare namespace cus = "http://schemas.devleap.com/cus";
for $c in doc("Cus.xml")/cus:customers/customer
where $c[@country='Italy']
order by $c[@name] descending
return
<customer name='{$c/@name}' city='{$c/@city}' />
```

Listing 3.1: Where Query

Here is the output from LINQ query: `Paolo Franco, Firenze`

And the output from XQuery: `<customer name="Paolo Franco" city="Firenze"/>`

3.2.2 `Select` Query

The following query in Listing 3.2 is a projection with an *index* argument in the *selector* predicate. It outputs as a result all customers, giving their index/ position, name and country.

```
LINQ:
var expr =
        customers
        .Select((c, index) => new {index, c.Name, c.Country});
foreach (var item in expr) {
        Console.WriteLine(item);
}

XQuery:
declare namespace cus = "http://schemas.devleap.com/cus";
for $c at $pos in doc("Cus.xml")/cus:customers/customer
return
<customer index='{$pos}' name='{$c/@name}' country='{$c/@country}'/>
```

Listing 3.2: Select Query

[3] `http://msdn.microsoft.com/en-us/magazine/cc337893.aspx`
[4] `http://rshelton.com/archive/2008/07/11/list-of-linq-providers.aspx`

Here is the output from LINQ query:

```
{ index = 0, Name = Paolo Franco, Country = Italy }
{ index = 1, Name = John Smith, Country = USA }
```

And the output from XQuery:

```
<customer index="1" name="Paolo Franco" country="Italy"/>
<customer index="2" name="John Smith" country="USA"/>
```

There is a difference at the indices returned by XQuery due to `$pos` function, which returns position values beginning at 1.

For a full listing of the queries that were included in eXist db performance test, please refer to the **Apendix** at the end of Chapter 3.

3.3 Appendix

SelectMany Query returns a list of `Quantity` and `IdProduct` of orders made by Italian customers.

```
LINQ:
var items = customers
        .Where(c => c.Country == Countries.Italy)
        .SelectMany(c => c.Orders,
         (c,o) => new { o.Shipped, o.IdProduct })

XQuery:
declare namespace cus = "http://schemas.devleap.com/cus";
for $c at $pos in doc("Cus.xml")/cus:customers
        /customer[@country='Italy']
let $o:=$c/order
return
<order shipped='{$o/@shipped}' idProduct='{$o/product/@id}'/>
```

Listing 3.3: SelectMany Query

LINQ Output: `Shipped: False - IdProduct: 1`

XQuery Output: `<order shipped="false" idProduct="1"/>`

```
LINQ:
var expr =
        customers
        .Where(c => c.Country == Countries.Italy)
        .OrderByDescending(c => c.Name)
        .ThenBy(c => c.City)
        .Select(c => new { c.Name, c.City })
        .Reverse();

XQuery:
declare namespace cus = "http://schemas.devleap.com/cus";
for $c in
reverse(doc("Cus.xml")/cus:customers/customer[@country='Italy'])
order by $c/@name descending,$c/@city
return
<customer name='{$c/@name}' city='{$c/@city}'/>
```

Listing 3.4: Reverse Operator Query

The **GroupBy** operator is used to group customer names by Country. As **GroupBy** is going to be implemented in XQuery version 1.1, currently it can be realized by using nested FLWORs in XQuery.

```
LINQ:
var expr = customers
        .GroupBy(c => c.Country, c => c.Name);
foreach (IGrouping<Countries, String> customerGroup in expr) {
         Console.WriteLine("Country: {0} - customerGroup.Key);
         foreach(var in customerGroup) {
                Console.WriteLine("\t{0}", item)
}

XQuery:
declare namespace cus = "http://schemas.devleap.com/cus";
for $c in distinct-values(doc("Cus.xml")//customer/@country)
let $cus:=doc("Cus.xml")//customer[@country=$c]
order by $c
return <country name='{$c}'>{
                for $cust in $cus
                order by $cust/@name
                return <customer name='{$cust/@name}' />
}</country>
```

Listing 3.5: WHERE Query

LINQ Output:

```
Key:  Italy - Count:  2
Key:  USA - Count:  2
```

XQuery Output:

```
<country name="Italy"> <customer name="Paolo Franco"/> <customer name="Marco
Polo"/> </country>
<country name="USA"> <customer name="James Joyce"/> <customer name="Frank
Smith"/> </country>
```

The Join Operator Query joins orders with their corresponding products.

```
LINQ:
var expr =
        customers
         .SelectMany(c => c.Orders)
         .Join( products,
                o => o.IdProduct,
                p => p.IdProduct,
                (o,p) => new {o.Month, o.Shipped, o.IdProduct, p.Price})
                    ;

XQuery:
declare namespace cus = "http://schemas.devleap.com/cus";
for $order in doc("Cus.xml")//order,
    $product in doc("Cus.xml")//product[@id=$order/@id]
return <product month='{$order/@month}' shipped='{$order/@shipped}'
                idProduct='{$product/@id}' price='{$product/price}'/>
```

Listing 3.6: Join Operator Query

LINQ Output:

```
Month = January, Shipped = False, IdProduct = 1, Price = 10
Month = May, Shipped = True, IdProduct = 2, Price = 20
```

XQuery Output:

```
<product month="1" shipped="false" idProduct="1" price="10.00"/>
<product month="5" shipped="true" idProduct="2" price="20.00"/>
```

```
1  LINQ:
2  var expr =
3          customers
4           .SelectMany(c => c.Orders)
5           .Join(products,
6                   o => o.IdProduct,
7                   p => p.IdProduct,
8                   (o,p) => p)
9           .Distinct();
10
11 XQuery:
12 declare namespace cus = "http://schemas.devleap.com/cus";
13 for $order in distinct-values(doc("Cus.xml")/cus:customers/customer/
       order/@id),
14     $product in doc("Cus.xml")//product[@id=$order]
15 return <product id='{$product/@id}'/>
```

Listing 3.7: Distinct Operator Query

Chapter 4

Benchmarking of eXist XML DBMS

Summary. *In the following chapter we present the native XML database eXist, give some ideas for performance improvement of eXist, and make a summary on the execution results when posting the queries, created in Chapter 3, to the database.*

4.1 eXist db Overview

eXist[Mei09] is a native XML database that uses a proprietary data store (B+ trees[1] and paged files[2]). It can be run as a standalone database server, as an embedded Java library, or in the servlet engine of a Web application. Documents are stored in a hierarchy of collections. Collections can contain child collections and do not constrain documents to any particular schema or document type.

eXist supports all of XQuery except for schema import/validation and wildcard searches on the following and preceding axes. Extensions to XQuery include full-text searches, calls to the XML:DB API[3] (such as to store query results in the database), executing dynamically constructed XQuery statements, applying XSLT stylesheets to a node, working with HTTP, and executing arbitrary Java methods. eXist ships with a number of user-written function modules, including modules for compression; date/time and math operations; working with images, files, and geospatial data; using HTTP, email, XSL-FO[4], and JNDI[5]; working with relational databases; and performing XML differencing[6]. Updates are supported through the extensions to XQuery and XUpdate[7]. eXist also provides partial support for XInclude[8] and XPointer[9].

[1] `http://baze.fri.uni-lj.si/dokumenti/B+%20Trees.pdf`
[2] `http://www.ba-horb.de/~pl/BS_Skript/node22.html`
[3] `http://xmldb-org.sourceforge.net/xapi/`
[4] `http://www.w3.org/Style/XSL/`
[5] `http://java.sun.com/products/jndi/`
[6] `http://www.cs.wisc.edu/niagara/papers/xdiff.pdf/`
[7] `http://xmldb-org.sourceforge.net/xupdate/`
[8] `http://www.w3.org/TR/xinclude/`
[9] `http://www.w3.org/TR/WD-xptr`

eXist supports the XML:DB API, with additional services for preparing and executing XQuery statements, managing users, managing multiple database instances, and querying indexes. DOM and SAX are supported for documents returned through the XML:DB API and live (read-only) DOM trees[10] are available when eXist is used as an embedded database.

eXist automatically indexes all element and attribute structure. By default, it creates full text indexes over all text and attribute values, but users can turn this off for selected parts of a document. It supports concurrent read/write access for multiple users. Security is provided through Unix style access permissions for both users and groups, which can be applied to both collections and individual documents. XQuery access control is supported through the eXtensible Access Control Markup Language (XACML).

4.2 Performance Optimization of eXist db

eXist db offers default indexing of the data collections, as well as a user-defined custom indexing. It allows users to take advantage of full-text, ranged, n-gram indexing[11], as well as the new Lucene-based index[12], used by eXist db version 1.3.

In order to achieve a better performance, some recommendation for the queries posted on eXist may be useful:

- **Short Paths are better**
  eXist uses indexes to directly locate an element or attribute by its name. It does not need to traverse the entire document tree. This means that the direct selection of a node through a single descendant step is faster than walking down the child axis. For example:

  ```
  a/b/c/d/e/f
  ```

  will be slower than

  ```
  a//f
  ```

  The first expression requires **six** index lookups while the second just needs **two**.

- **Allow eXist to process large node sets in one step**
  The query engine is optimized to process a path expression in one single operation. Unlike most XQuery engines, which operate on in-memory documents, eXist tries to avoid tree traversals and prefers index-based node set operations wherever possible. Owing to eXist's indexing, the XPath expression:

  ```
  //A/*[B = 'C']
  ```

  can be evaluated in a single operation for all context items. It doesn't make a difference if the input set comes from a single large document, includes all the documents in a specific collection, or even the entire database. The logic

[10] http://www.w3.org/TR/DOM-Level-2-Core/introduction.html
[11] http://www.cs.umbc.edu/ngram/
[12] http://lucene.apache.org/java/docs/

of the operation remains the same. However, many queries can force the query engine to partition the input sequence and process it in an item-by-item mode. Several examples for "bad" uses of FLWOR expressions will be given below. For example, most function calls will also force the query engine into item-by-item mode:

```
//A/*[f:process(B) = 'C']
```

The function has to be called once for every instance of B. Normally, eXist would try to evaluate the general comparison in a single step, assuming there is a usable index on B. However, it now needs to call a (non-optimized) function for each B and will thus need to process the entire comparison once for every context item. Users therefore need to be aware of the cost when using funcion calls to process huge node sets.

There are functions to which the above does not apply. This includes most functions which operate on indexes e.g., `contains`, `matches`, `starts-with`, `ngram:contains`, and the like. They are optimized so eXist only needs to call them once to process the entire context set. For example, using `ngram:contains` as below is correct:

```
//A/*[ngram:contains(B, 'C')]
```

while

```
//A/*[ngram:contains(f:process(B), 'C')]
```

will again force eXist into step-by-step evaluation.

- **Prefer XPath Predicates Over Where Expressions**
  This is a variation of the problems discussed above. Many users tend to formulate SQL-style queries using an explite `where` clause, for example:

  ```
  for $e in //entry where $e/@type = 'subject' return $e
  ```

  could be rewritten as the equivalent query using XPath predicate:

  ```
  for $e in //entry[@type = 'subject'] return $e
  ```

  The `for ...where` expression forces the query engine into a step-by-step iteration over the input sequence, testing each instance of `$e` against the where expression and possible optimizations are lost.

  Contrary to this, the XPath predicate expression can be processed in one single step, making best use of any available indexes. There are use cases which cannot be handled without using `where` e.g., `joins` between multiple documents. However, users shouldn't use `where` if they can replace it by a simple XPath.

4.3 Query Results

The execution time of the following query from Listing 3.7 can be seen below:

```
declare namespace cus = "http://schemas.devleap.com/cus";
for $c in doc("Cus.xml")/cus:customers/customer
where $c[@country='Italy']
order by $c[@name] descending
return
<customer name='{$c/@name}' city='{$c/@city}' />
```

Listing 4.1: Where Query

```
Found 168 in 0.297 seconds.
```

However, if the query is written by replacing `where` clause with an XPath expression, in the form:

```
declare namespace cus = "http://schemas.devleap.com/cus";
for $c in doc("Cus.xml")/cus:customers/customer[@country='Italy']
order by $c[@name] descending
return
<customer name='{$c/@name}' city='{$c/@city}' />
```

Listing 4.2: Where Query transformed as an XPath expression

the results are fetched for: `Found 168 in 0.062 seconds.`

For a summarized data on all queries results, please refer to the table below:

eXist db Performance		
Query	**Returned**	**Execution**
Where	168	0.297 sec.
Where/Xpath	168	0.062 sec.
Select	1000	0.109 sec.
SelectMany	168	0.062 sec.
Reverse	168	0.046 sec.
Join	1009	18.328 sec.
GroupBy	3	0.141 sec.
Distinct	1007	8.422 sec.

Table 4.1: Queries Results

In 2006 W3C conducted a large test suite on 16 XQuery engines, including eXist db. The results are publihed on W3C's web-page[13]. From the summary can be seen that the performance of eXist db was in the 'golden middle', as compared to the other 15 XQuery engines that were tested. eXist's conformance to the test suite was 99.4%.

In 2009, the new eXist version 1.3 already provides the missing functionality from 2006, with the exeption of *Schema Validation Feature*. The open source eXist db project is a well-documented, reliable, and free XQuery engine and native XML database, with an active mailing list to support users, and a team of contributors, that work for its further development.

[13] `http://www.w3.org/XML/Query/test-suite/XQTSReport.html#testRun`

Chapter 5

Conclusion and Future Works

Summary. *In this chapter, we give the conclusion and offer some ideas that are probable candidates for future work.*

5.1 Conclusion

The objectives of this project work were to present EMF Eclipse framework as a model generator, and to perform an evaluation of eXist db performance, used as a model repository for EMF Objects. The second target was to present the fuctional query languages LINQ and XQuery, and to use them for the process of XML database performance testing.

After introducing the topic in Chapter 1, Chapter 2 provides details on model generation, intantiation, and serialization, by using Eclipse EMF. In Chapter 3, is offered a translation of LINQ to XQuery queries, followed by Chapter 4, where some features of eXist are offered, as well as several suggestions for performance optimization. Chapter 4 also presents the results of eXist db performance tests.

5.2 Future Works

EMF Objects can be manipulated and retrieved using queries. The queries can be evaluated on client side or they can be moved to server side for evaluation (query shipping) for large repositories. In this case, the query language (LINQ, OCL) is translated into the query language of the database engine. In order to avoid the additional query translation, the performance of an eXist DBMS was evaluated, used as a model repository for EMF Objects. The query language XQuery was used for querying the database.

There is a Java Specification Request[1] to define a set of interfaces and classes that enable an application to submit XQuery queries to an XML data source and process the results of these queries. As XML is a universally used format for data exchange used in XML DBMSs, and XQuery is the XML Query language supported by W3C, a future work may be the development of an XQuery for Java provider, that enables the direct querying of XML data in large XML data repositories, from Java applications,

[1] `http://jcp.org/en/jsr/detail?id=225`

in order to provide integrated query functionality to Java.

Another direction, in which future work may be done, is to provide optimizations for such integrated query functionality. Users should be able to write native query expressions and the database should execute them with performance equal to the string-based queries. The inspiration for developing a query optimizer as part of future work comes from the University of Texas project, titled lambda-DB[2]. This project aims at developing frameworks and prototype systems that address the query optimization challenges for OODBs.

[2] `http://lambda.uta.edu/lambda-DB/manual/overview.html`

Bibliography

[Ams] CWI Amsterdam. Xmark homepage. `http://monetdb.cwi.nl/xml/index.html`.

[AS01] Martin Kersten Daniela Florescu Albrecht Schmidt, Florian Waas. Why And How To Benchmark XML Databases. In *CWI, Microsoft, Propel*, 2001.

[Ben93] *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann Publishers, 1993.

[Bou05] Ronald Bourret. Xml and databases. *www.rpbourret.com*, 2005.

[EMF08] *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Longman, 2008.

[Gui] Eclipse EMF Developer Guide. Generating an emf model using xml schema (xsd). `http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/tutorials/slibmod/slibmod.html`.

[Lei] University Leipzig. Xmach-1: A benchmark for xml data management. `http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html`.

[Lin08] *Programming Microsoft LINQ.* Microsoft Press, 2008.

[Mei09] Wolfgang Meier. exist xml db, 2009. `http://www.exist-db.org/`.

[NM08] Anders Norås and Thomas Mueller. Quaere, 2008. `http://quaere.codehaus.org/`.

[OV08] Audris Kalnins Oskars Vilitis. A Proxy Approach to External Model Repository Integration in Eclipse EMF Infrastructure. In *Institute of Mathematics and Computer Science, University of Latvia*, June 2008.

[Pri08] Rakesh Prithiviraj. Ide customization to support language embeddings. Master's thesis, TUHH, 2008.

[Red04] *XML for DB2 Information Integration.* IBM,Redbooks, 2004.

[Sin] National University Singapore. The xoo7 bechmark. `http://www.comp.nus.edu.sg/~ebh/XOO7.html`.

[Tri08] Konstantin Triger. Jaque, 2008. `http://code.google.com/p/jaque/`.

[Wes09] Timo Westkämper. Querydsl, 2009. `http://source.mysema.com/display/querydsl/Querydsl`.