Data Aggregation, Heterogeneous Data Sources and Streaming Processing: How Can XQuery Help?

Marc Van Cappellen, Wouter Cordewiner, Carlo Innocenti
XML Products, DataDirect Technologies

Abstract

Software infrastructures and applications more and more must deal with data available in a variety of different storage engines, accessible through a multitude of protocols and interfaces; and it is common that the size of the data involved requires streaming-based processing.

This article shows how XQuery can leverage the XML Data Model to abstract the data physical details and to offer optimized processing allowing the development of highly scalable and performant data integration solutions.

1 Introduction

Data access has always been a hot topic. The variety of interfaces available for querying, creating and updating data is impressive and constantly growing. JDBC, ODBC, ADO .NET are the typical basic interfaces you will deal with when working with relational data sources; but don't ignore also Object Relational Mapping systems, like Hibernate [4] for example. If you need to deal with XML, you will most likely hear about DOM, SAX and StAX interfaces; or maybe object to XML mapping, like JAXB [3], for example. Things get even more difficult when dealing with different data formats, like Electronic Data Interchange messages (EDI) or even flat files. The choice about which data access solution to use in which scenario becomes more complicated when your application needs to deal with multiple, physically varied data sources, which is a typical problem especially when dealing with SOA [1].

SOA (Service Oriented Architecture) [8] has been around for a number of years earning acceptance as a solid approach for systems management - one that allows for the broad reuse of existing software assets, provides a sound architectural model for the federation of disparate IT systems, and supports the automation of abstract business processes via a range of programming paradigms.

But how does data management fit in? Guidelines for service-oriented data access and management techniques are sparse. Those that are available have typically been formulated by SOA experts, not data management experts. As a result, different understandings of the same problems turn into a constant source of confusion and headaches.

Most SOA data management solutions currently in use rely on traditional, well defined interfaces including ODBC, JDBC, OCI, ADO.NET, OLE DB, and others. All of these interfaces share similar concepts, but most of them fail to capture the differences between traditional data access architecture characteristics (tightly coupled,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

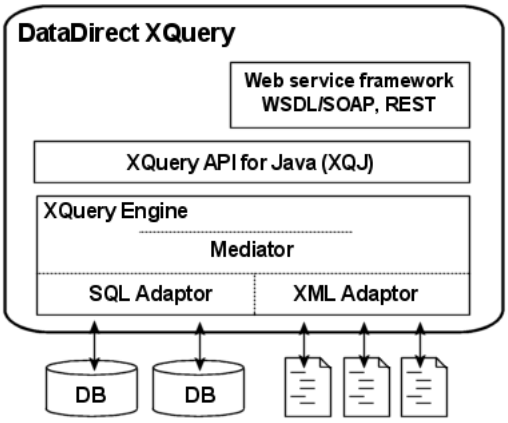

Figure 1: Overview of the DataDirect XQuery engine architecture

complex state machine, connection based, and relational model driven) and characteristics associated with SOA (loosely coupled, stateless, message-centric, and typically XML-based data interchange).

XQuery [10], the XQuery for Java API [6], and Web services [9] provide a great way to bridge the data disparity with service-orientation. XQuery still exposes an interface against which users submit queries and from which they process results, but at the same time it is easily embedded in an application or exposed via a Web service, and it further provides abstraction between the consumer of the data and the physical details about how the data is stored. XQuery is designed to give language implementations the possibility to execute queries against heterogeneous data sources, interpreting (but not necessarily materializing) all of them as XML.

XQuery is based on an XML data model, providing smooth integration in todays Web service-centric infrastructures. When you consider the service-orientation design paradigm, it becomes evident that XQuery features are very much in alignment with the goals of service-oriented computing.

Making XQuery work efficiently against heterogeneous data sources presents peculiar challenges. DataDirect XQuery is an XQuery implementation that was first released in year 2005. DataDirect XQuery's design emphasizes performance and scalability across heterogeneous data sources, with a strong focus on relational data and large XML documents. This paper reviews some of the techniques DataDirect XQuery uses to optimize access to relational and XML data sources.

2 Background

Figure 1 describes the high level architecture of the DataDirect XQuery engine. The engine is accessed through either a Web service framework [2], which allows easy deployment with most application servers, or through a standard API, the XQuery API for Java (XQJ) [6]; that's the interface that can be used to access DataDirect XQuery directly from a client application. The XQuery engine itself is split in three logical components:

- Mediator
  The Mediator is the component that takes care of decomposing an XQuery based on which data sources are being accessed, and of merging the result back from the various data sources.

- SQL adaptor
  The SQL adaptor is responsible for handling the parts of the query that are dealing with relational databases and for pushing the burden of the query to the database itself.

- XML adaptor
  The XML adaptor is responsible for handling the parts of the query that are dealing with XML data

sources, implementing a variety of optimization techniques that allow the process to be highly scalable and performant.

Each of these logical components covers a critical role in making the DataDirect XQuery engine able to deal with heterogeneous data sources. There are several general optimization tasks that are accomplished by the Mediator and that you can probably find described in the context of various XQuery implementations; but some of the most sophisticated optimizations occur at the data source adaptor level. The following sections focus on the specific techniques implemented by the SQL and XML adaptor.

3 XQuery and relational data sources

XQuery and SQL support different operations on very different data models; XQuery works against the XQuery Data Model (XDM) [11], while SQL works against a Relational Data Model; XQuery is designed to make navigation of XML hierarchical structures easy, while SQL focuses more on the task of joining multiple related tables and creating projections on the result. And that's just scratching the surface in terms of differences.

Some XQuery implementations just materialize entire relational tables as XML; others issue the same SQL regardless of the database involved; others yet rely on the least common denominator functionality of the least capable JDBC drivers, which limits performance significantly; and many perform most XQuery functions in the XQuery engine instead of evaluating them in the database.

DataDirect XQuery has been designed to optimize performance and scalability when dealing with all its supported data sources, especially relational databases and large XML documents. Before taking a look at specific XQuery examples and how DataDirect XQuery executes them, let's take a look at the main high level techniques implemented by the SQL adaptor:

- Minimal data retrieval
  Moving data is expensive. In DataDirect XQuery, the generated SQL is as selective as possible, retrieving only the data needed to create the results of a query. It is not unusual that in some cases DataDirect XQuery fetches only part of a single row where other XQuery implementations return an entire table

- Leverage the database
  DataDirect XQuery pushes down into the database operations that can be performed in SQL; that way the relational query optimizer can leverage indexes and other structures. The performance gains this brings are particularly important for joins, Where and Order By clauses, and SQL functions. This also reduces data retrieval, since data need not be retrieved for operations to be done in the database.

- Optimize for each database
  Today's relational databases support significantly different dialects of SQL, and even when two databases support the same operation, their performance may be quite different. Most databases have enough functionality to support XQuery efficiently, but the constructs needed to do this are different for each database. Some XQuery implementations support only one database; others generate the same SQL regardless of the database involved, which results in poor performance. In contrast, DataDirect XQuery uses a different SQL adaptor for each database, generating SQL specifically optimized for that database.

- Support incremental evaluation
  In many applications, results are returned to the user as soon as they are available, displaying the first results well before the entire query has been performed. Many XML applications are based on streaming architectures. DataDirect XQuery uses lazy evaluation so that streaming APIs can retrieve data as soon as it is available. As data is needed, the engine retrieves it incrementally from JDBC result sets. Because there is no need to have the entire result in memory at one time, very large documents can be created.

- Optimize for XML hierarchies
  Because XML construction is hierarchical, DataDirect XQuery uses SQL algorithms that optimize retrieving data for building hierarchies. For instance, the XQuery engine makes extensive use of merge-joins when building hierarchical documents.

- Give the programmer the last word
  Every SQL programmer knows that occasionally hints are needed to get optimal performance for a specific query. This is also true in XQuery, so DataDirect XQuery allows programmers to influence the SQL it generates. This can significantly improve performance in some cases.

The following sections illustrate examples of how XQueries defined against relational databases are translated into SQL by the DataDirect XQuery's SQL Adaptor. Most of the generated SQL shown in this paper is for Oracle 11g - SQL generated for other databases may look significantly different.

The following examples all assume a simple database structure made of two tables, *HOLDINGS* and *USERS*, which contain information about how many and what kind of stock holdings users of the system own.

3.1 Querying data

To minimize data retrieval, DataDirect XQuery generates very selective SQL, returning only the data that is needed for a given XQuery. To avoid retrieving rows that are not needed, the conditions in Where clauses and predicates are converted to Where clauses in the generated SQL. To avoid retrieving columns that are not needed, the generated SQL specifies the columns actually needed to evaluate the XQuery.

3.1.1 Where clause pushdown

Consider the following XQuery, which retrieves all *holdings* for less than 10,000 shares; the XQuery can be easily written in two different ways, one using the Where clause, the other using straight XPath predicates.

```
for $h in collection('HOLDINGS')/HOLDINGS
where $h/SHARES < 10000
return $h
```

```
collection('HOLDINGS')/HOLDINGS[SHARES < 10000]
```

For both XQueries, the SQL query generated by DataDirect XQuery fetches and returns only the rows that are actually to compose the XQuery result:

```
SELECT ALL nrm4."USERID" AS RACOL1, nrm4."SHARES" AS RACOL2, nrm4."STOCKTICKER" AS RACOL3
FROM "MYDB"."HOLDINGS" nrm4
WHERE nrm4."SHARES" < 10000
```

3.1.2 Projection pushdown

The following XQuery retrieves first and last name for each user older than 40. The XQuery is similar to the example shown in the previous section, but this time instead of returning the whole row meeting the selection criteria, the query only needs to retrieve two fields:

```
for $user in collection('USERS')/USERS
where $user/AGE > 40
return <user>{$user/FIRSTNAME, $user/LASTNAME}</user>
```

In this case the DataDirect XQuery engine needs to push to the SQL engine the Where clause, and the selection of two specific columns:

```
SELECT ALL nrm5."FIRSTNAME" AS RACOL1, nrm5."LASTNAME" AS RACOL2
FROM "MYDB"."USERS" nrm5
WHERE nrm5."AGE" > 40
```

3.1.3 Join pushdown

Relational databases are designed to optimize joins, so DataDirect XQuery leverages the database when an XQuery join involves SQL data. Performing all the joins in the database typically results in a dramatic performance gain.

Consider the following XQuery, which retrieve all users and stock holdings for each user:

```
for $u in collection('USERS')/USERS,
    $h in collection('HOLDINGS')/HOLDINGS
where $u/USERID = $h/USERID
return <holding name="{$u/LASTNAME}">{$h/SHARES/text()}</holding>
```

The SQL generated by DataDirect XQuery pushes the resolution of the join operation to the database:

```
SELECT ALL nrm5."LASTNAME" AS RACOL1, nrm9."SHARES" AS RACOL2
FROM "MYDB"."USERS" nrm5, "MYDB"."HOLDINGS" nrm9
WHERE nrm5."USERID" = nrm9."USERID"
```

There are of course multiple ways to express the same join condition in XQuery; for example, in this case the same condition could have been expressed using an XPath predicate, like in:

```
for $u in collection('USERS')/USERS, $h in collection('HOLDINGS')/HOLDINGS[USERID = $u/USERID] return ...
```

DataDirect XQuery is able to capture the multiple ways to express the same queries and it will push down the same SQL.

4 XQuery and XML data sources

While optimizing XQuery when working against relational data sources is mostly a matter of issuing the *best* SQL to the server and to lazily fetch results, when querying XML data sources the XQuery engine needs to deal with the physical task of analyzing and filtering the data. XML data sources include:

- XML documents
- Web service call results (typically SOAP responses)
- Office Open XML (OOXML) or OpenDocument format (ODF) documents
- Comma Separated Value (CSV) files, Tab Delimited files or other flat file formats
- Electronic Data Interchange (EDI) messages streamed to XML

Software architects often tend to underestimate the challenges offered by querying XML documents; that's why often that operation becomes the bottleneck of complex systems. What may start as an application designed to deal with a few relatively small XML documents can easily need to scale up to handle hundreds of XML documents per second, or XML documents that grow to be several Gigabytes in size.

DataDirect XQuery has been optimized to handle data sources in a highly scalable and performant way. The engine's XML adaptor implements several techniques to accomplish that task, like general execution tree optimizations (including function inlining, detecting loop invariants, etc.), in-memory indexing and more; but two major techniques stand out: document projection and streaming processing.

4.1 XML document projection

XML document projection is a clever idea introduced originally by Amelie Marian and Jerome Simeon [7]. The basic idea behind document projection is: given an XML document that represents several details for each *item*, if my XQuery only needs to retrieve/query a couple of attributes for each *item*, why should the XQuery engine materialize in memory the whole *item* elements?

Consider this simple XML document, describing a few objects available for auction:

```
items.xml:
  <ITEMS>
    <ITEM>
      <ITEMNO>1002</ITEMNO>
      <DESCRIPTION>Motorcycle</DESCRIPTION>
      <OFFERED_BY>U02</OFFERED_BY>
      <START_DATE>1999-02-11T00:00:00</START_DATE>
      <END_DATE>1999-03-25T00:00:00</END_DATE>
      <RESERVE_PRICE>500</RESERVE_PRICE>
    </ITEM>
    <ITEM>
      <ITEMNO>1003</ITEMNO>
      <DESCRIPTION>Bicycle</DESCRIPTION>
      <OFFERED_BY>U02</OFFERED_BY>
      <START_DATE>1999-02-11T00:00:00</START_DATE>
      <END_DATE>1999-03-25T00:00:00</END_DATE>
      <RESERVE_PRICE>200</RESERVE_PRICE>
    </ITEM>
  </ITEMS>
```

And now consider this XQuery that retrieves the auction end date for a specific *ITEM* in the XML document above:

```
  for $i in doc('items.xml')/ITEMS/ITEM
  where $i/ITEMNO eq '1002'
  return $i/END_DATE
```

The XQuery only needs two pieces of information for each *ITEM* in the source XML document: *ITEMNO* to resolve the search criteria and *END_DATE* to return the required result. The only parts of the input XML document that are instantiated in memory are the ones highlighted in the following XML fragment:

```
<ITEMS>
  <ITEM>
    <ITEMNO>1002</ITEMNO>
    <DESCRIPTION>Motorcycle</DESCRIPTION>
    <OFFERED_BY>U02</OFFERED_BY>
    <START_DATE>1999-02-11T00:00:00</START_DATE>
    <END_DATE>1999-03-25T00:00:00</END_DATE>
    <RESERVE_PRICE>500</RESERVE_PRICE>
  </ITEM>
  <ITEM>
    <ITEMNO>1003</ITEMNO>
    <DESCRIPTION>Bicycle</DESCRIPTION>
    <OFFERED_BY>U02</OFFERED_BY>
    <START_DATE>1999-02-11T00:00:00</START_DATE>
    <END_DATE>1999-03-25T00:00:00</END_DATE>
    <RESERVE_PRICE>200</RESERVE_PRICE>
  </ITEM>
</ITEMS>
```

DataDirect XQuery statically analyzes an XQuery and generates a *projection tree*; in the example above, the projection tree can be expressed as:

```
  +-step axis="self" test="document-node()"
    +-step axis="child" test="ITEMS"
      +-step axis="child" test="ITEM"
        +-step axis="child" test="ITEMNO"
          +-step axis="descendant" test="node()"
        +-step axis="child" test="END_DATE"
          +-step axis="descendant" test="node()"
```

The projection tree is used in DataDirect XQuery as part of the content handler which processes the XML parser events, ensuring that only the necessary XML parts included in the projection tree are actually materialized in memory.

The process is typically not as simple as the one described in the example above; just think, for example, about the necessary steps needed to handle expressions like *//ITEM* (path reduction) or *../ITEM* (parent axis). But the benefits in terms of performance and scalability when dealing with large XML documents are often impressive, even when *document streaming* (described below) is not available.

4.2 XML document streaming

Processing XQuery in streaming fashion is the ultimate solution in terms of querying XML documents in a scalable way. In the ideal case, when running an XQuery against one (or more) XML document(s) in streaming mode the amount of memory required by the XQuery engine doesn't grow proportionally to the size of the input(s). That allows XQuery to run against XML documents much larger than the physical memory available on a workstation, even when XML document projection can't help.

Consider an XML document similar to the one discussed above in the context of XML document projection, where this time the number of listed *ITEM* elements is in the order of millions. When DataDirect XQuery analyzes the following XQuery, it creates the projection tree and it knows it can avoid materializing in memory several sub-elements for each analyzed *ITEM* element:

```
<MYITEMS> {
  for $i in doc('items.xml')/ITEMS/ITEM
  where $i/OFFERED_BY eq 'U02'
  return
    <ITEM>{$i/ITEMNO, $i/DESCRIPTION, $i/RESERVE_PRICE}</ITEM>
} </MYITEMS>
```

But still, there is a large amount of information that would need to be stored in memory to execute the XQuery in a traditional manner (with no streaming processing); and the amount of required memory would indeed be proportional to the size of the input XML document. Thanks to the XML document streaming technique, DataDirect XQuery is able to process the XQuery described above in streaming fashion, which means that only one *ITEM* per time is actually materialized in memory and discarded when no more needed.

It's worth noting that XML document projection and streaming are two complementary implementation techniques, which implies that when an XQuery is processed in streaming fashion, XML document projection still takes place, limiting the amount of data temporarily materialized by the streaming engine.

When document streaming is used in conjunction with one of the streaming interfaces to consume the result, like StAX [5] for example, which is supported by the XQuery API for Java standard, the whole XQuery processing works in a purely streaming fashion, with the XQuery engine consuming parts of the input XML document on demand based on the way the client application is consuming the XQuery result.

Thanks to these XQuery processing techniques, applications are able to process XML documents in the range of several dozens of Gigabytes without incurring in scalability issues.

5 Mixing data sources together

In the previous sections we have discussed several techniques implemented by DataDirect XQuery to optimize processing of XQuery when working against relational or XML data sources. But it is common that applications need to deal with data which is not available in a single format; and that's the context where dealing with a single query language, data model and interface which covers heterogeneous data sources becomes fundamental.

Think about a scenario, for example, where a list of auctioned *ITEM*s is available in an XML document, as described in 4.1, but details about the person who's offering the *ITEM* are available in a *USERS* table hosted on a relational database, including information about the user id, name, address and email. Now think about the need of creating an application that given a user's email address retrieves all the items that are being auctioned by that user.

Thanks to XQuery, that task can be solved by a single, simple query. Note how the XQuery author doesn't need to worry about different data models or different interfaces to the underlying physical data sources:

```
<ITEMS> {
  let $user := collection("USERS")/USERS[USERID = "U02"]
  for $i in doc('items.xml')/ITEMS/ITEM
  where $i/OFFERED_BY eq $user/USERID
  return
    <ITEM>{$user/NAME, $i/ITEMNO, $i/DESCRIPTION, $i/RESERVE_PRICE}</ITEM>
} </ITEMS>
```

Thanks to the optimization techniques discussed above about how DataDirect XQuery handles relational and XML data sources, the query above will take full advantage of the performance capabilities of the database engine hosting the *USERS* table, and of the document projection and streaming processing in dealing with the *items.xml* XML document.

The application consuming the result is shielded from the physical origin of the data returned by the XQuery; even if the result mixes information stored in a relational database and in an XML document, the client application doesn't need to know about that, and it is able to access the returned data through the standard interfaces exposed by the XQuery API for Java.

6 Conclusions

In this paper we have discussed how XQuery can be useful in providing data services which accomplish data integration tasks across heterogeneous data sources. In order to succeed in that task, XQuery implementations must be optimized to deal with the peculiarities of the various supported data sources. DataDirect XQuery implements a variety of techniques when dealing with relational databases and XML documents; those include the ability to push SQL to the relational engine, to minimize the amount of data retrieved from the database, to leverage XML document projection and XML document streaming to handle large XML documents in an efficient and scalable way. Thanks to these techniques XQuery is an excellent technology for simplifying and streamlining data access in the context of traditional and SOA applications.

References

[1] Jason Bloomberg and John Goodson. Best Practices for SOA: Building a Data Services Layer. *SOA World Magazine*, May, 2008.

[2] DataDirect Technologies. DataDirect XQuery Web Service Framework. *http://www.xquery.com/examples/web-service-example/xquerywebservice/*.

[3] S. Vajjhala and J. Fialli. The Java architecture for XML binding (JAXB) 2.0. *http://jcp.org/en/jsr/detail?id=222*.

[4] Red Hat Middleware, LLC. Java Persistance with Hibernate. *http://www.hibernate.org/397.html*.

[5] Java Community Process. JSR 173: Streaming API for XML (StAX). *http://jcp.org/en/jsr/detail?id=173*.

[6] Java Community Process. JSR 225: XQuery API for Java (XQJ). *http://jcp.org/en/jsr/detail?id=225*.

[7] A. Marian and J. Simeon. Projecting XML Documents *Bell Laboratories*, France, 2003.

[8] M. Huhns and M. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 1(9):75–81, 2005.

[9] WWW Consortium. Web Services. *http://www.w3.org/2002/ws/*.

[10] WWW Consortium. XQuery 1.0: An XML Query Language. *W3C Recommendation*, 23 Jan 2007.

[11] WWW Consortium. XQuery 1.0 and XPath 2.0 Data Model (XDM). *W3C Recommendation*, 23 Jan 2007.