11-2014

XQuery for Archivists: Understanding EAD Finding Aids as Data

Gregory Wiedeman
University at Albany, State University of New York, gwiedeman@albany.edu

Recommended Citation
Wiedeman, Gregory. "XQuery for Archivists: Understanding EAD Finding Aids as Data." Practical Technology for Archives 3 (November 2014)

XQuery for Archivists: Understanding EAD Finding Aids as Data

Gregory Wiedeman

Abstract

XML has long been an important tool for archivists. The addition of XQuery provides a simple and easy-to-learn tool to extract, transform, and manipulate the large amounts of XML data that archival repositories have committed resources to develop and maintain – particularly EAD finding aids. XQuery allows archivists to make use of that data. Furthermore, using XQuery to *query* EAD finding aids, rather than merely reformat them with XSLT, forces archivists to look at finding aids as data. This will provide better knowledge of how EAD may be used and further understanding of how finding aids may be better encoded. This article provides a simple how-to guide to get archivists to start experimenting with XQuery.

XQuery is a simple, yet powerful, scripting language designed to enable users without formal programming training to extract, transform, and manipulate XML data. Moreover, the language is an accepted standard and a W3C recommendation much like its sister standards, XML and XSLT. In other words, XQuery's *raison d'etre* coincides perfectly with the needs of today's archivists. What follows is a brief, pragmatic, overview of XQuery for archivists that will enable archivists with a keen understanding of XML, XPath, and EAD to begin experimenting with manipulating EAD data using XQuery.

Archivists do not need to be sold on XML – repositories around the country have been early adopters to the technology since the 1990s. Essentially all archives data standards have been either developed for XML (EAD) or been adapted for its use (MARC). The openness, standardization, structural flexibility, and ease of use have proven that XML is a powerful and central tool for the 21st century archivist. Yet, while archivists have long stored finding aids in EAD, there is a major skill difference between encoding finding aids and actually doing anything with that data.[1] The most typical way archivists have made use of XML data is through XSLT stylesheets, often to display data in long, scrollable lists. More recently, there has been a movement to develop more sophisticated and user-friendly archival information systems.[2] Furthermore, archives that have so heavily invested in developing EAD finding aids would love to automate the repurposing this data for easier access, outreach activities, and more. Quite simply, manipulating and reformatting XML data has become a valuable skill for archivists and XQuery provides a simple and easy-to-learn method to do just that.

This is by no means even a comprehensive introduction to XQuery – for that, archivists will need to seek more traditional resources like those supplied at the end of this article. Yet, starting with a long and comprehensive study of XQuery may serve only to delay hands-on experience and frustrate the impatient. Archivists may find it easier to dive in and experiment with the language before seeking a broader understanding. What archivists need is a simple and accessible guide to get them started. If you run into trouble while using this guide, try searching for your problem either at Stack Overflow or with your favorite search engine.[3]

XQuery and XSLT

Now, to archivists who have had some experience with XSLT, XQuery sounds a lot like these XML stylesheet transformations. XSLT can also be used to transform and manipulate XML data, was put into widespread use well before XQuery, and has the advantage of being compiled by web browsers. This means XML data can be processed server-side with XSLT whereas today's browsers are unable to read XQuery without specialized add-ons or workarounds. Archivists have not had to learn and adapt to new

[1] Sonia Yaco, "It's Complicated: Barriers to EAD Implementation," *The American Archivist* 71 (Fall/Winter 2008):467.
[2] J. Gordon Daines & Cory L. Nimer, "Re-Imagining Archival Display: Creating User-Friendly Finding Aids," *Journal of Archival Organization* 9 (2011): 4-31. ArchivesSpace: http://www.archivesspace.org. Princeton University Library Finding Aids: http://findingaids.princeton.edu.
[3] Stack Overflow: http://www.stackoverflow.com.

software to try XSLT – they have just experimented with stylesheet transformations by using their familiar web browser. This may be why XSLT has enjoyed much more use within the archives community. Graduate schools have commonly taught XSLT and many archivists have spent a considerable amount of time learning the standard. Yet, while XSLT and XQuery do overlap in some ways, there are a few important distinctions that make the latter more useful in many cases.

Why XQuery?

So why use or learn XQuery? Most importantly, it is much simpler and less verbose then XSLT, which is written in XML itself. This makes XSLT require more characters to perform the same actions than XQuery (see appendix for comparison). While this does not seem like a huge advantage, it actually is. XQuery scripts are cleaner, simpler, often quicker to write, and much easier to maintain. This makes users *much* more likely to actually use the language and make more effective use of their XML data. As Steve Krug famously argued about usability testing: if a task is difficult it will be avoided, while if a task is easier the more likely that it will be performed more often.[4] Archivists who have avoided updating large XSLT files derived long ago from the original EAD cookbook will certainly sympathize.[5]

Secondly, XQuery is more powerful than XSLT. It can perform more functions and make complex tasks easier. Functions are central to both languages – think of them as pre-programmed magic words that help users to easily perform complex actions with their data. In XQuery, advanced users can even write their own functions more easily than in XSLT. While XSLT 3.0 has introduced more functions, this does not even compare to XQuery which has 225 built-in functions that can easily check if an element exists or contains data, edit character strings in complex ways, determine the relative positions of elements, and many more tools that enable users to get the most out of their data.

[4] Steve Krug, *Don't Make Me Think: A Common Sense Approach to Web Usability* (Berkeley, CA: New Riders Publishing, 2006).
[5] Michael J. Fox, *The EAD Cookbook* (2000), http://saa-ead-roundtable.github.io/.

So, XQuery enables archivists to do more with their data, and makes them much more likely to do it. Yet, the biggest advantage might be that it forces them to think of EAD files as *data* not as a list or index. Unlike XSLT, XQuery is designed to *query* XML – it is designed so that users ask what data they want and how they want it. This will force archivists to see their descriptions as discrete units of information. Not only do descriptions have contextual relationships with surrounding descriptions that convey original order, but it can also be useful to return discrete descriptions as search results or reorder data to show information in different (and perhaps more accessible) ways. This is consistent with recent arguments that single levels of EAD finding aids are useful as discrete elements for navigation and will overall serve to further distance EAD from the display that researchers see.[6] Thus, EAD will become a data store of archival description while an additional interface or information system *queries* it rather than transforming or reformatting it.

What Do I Need to Use XQuery?

While XQuery cannot run natively in common bowsers, there are a number of ways to begin experimenting with querying XML. First, Saxon processes XQuery as well as XSLT, so if you have access to desktop software such as Oxygen XML Editor or a server that is running Saxon, you should be able to run queries much like you run XSLT.[7] Saxon can also be set up to run on a desktop using the Java Runtime Environment though the command line. Additionally, eXist-db is an open source XML database built specifically to run XQuery with a RESTful architecture – essentially, eXist is meant to be part of the back end of a typical web service.[8] Here the software runs on a server (or is simulated using the Java Runtime Environment) and users log on to a web interface to upload and manage XML data. XQuery scripts can then be run within web pages by end-users. Since HTML is XML itself, with eXist XQuery can

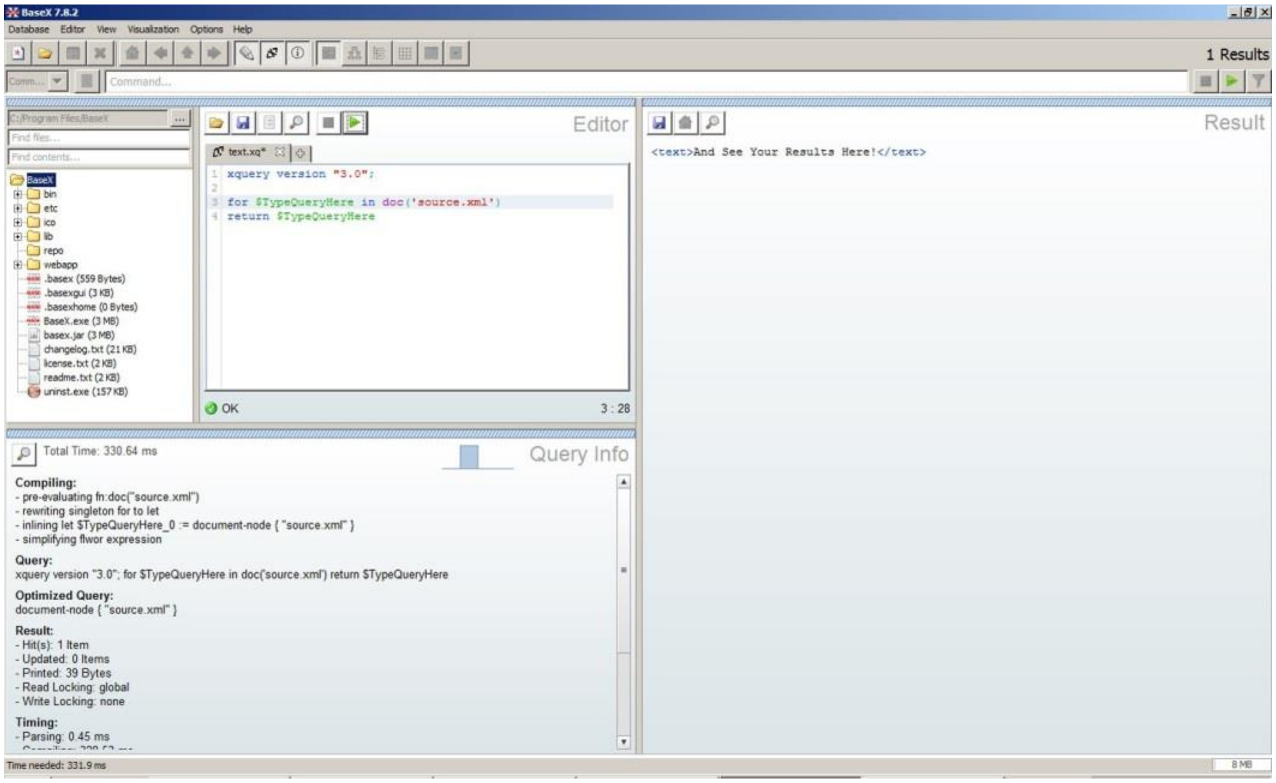
[6] Daines & Nimer, 13-20.
[7] SAXON: The XSLT and XQuery Processor: http://saxon.sourceforge.net. Some variations of Saxon only run XQuery 1.0, while others can process the additional functions in XQuery 3.0.
[8] eXist-db: The Open Source Native XML Database: http://exist-db.org/exist/apps/homepage/index.html.

edit, transform and query HTML and function much like a server-side scripting language. The Princeton University Library Finding Aids make use of eXist-db. Finally, Zorba is an XQuery processor that can run in the command line or as an extension to PHP or Python. [9] This enables XQuery to be run within server-side PHP or Python web applications.

The above methods for running XQuery scripts might seem scary or confusing to many archivists, particularly those with little experience with servers and web architecture. This, compared to native browser support, seems to be one of the reasons XSLT has remained popular. However, there are also easier and more accessible ways to experiment with XQuery. After all, it is just text.

Perhaps the easiest way to begin playing with XQuery is using BaseX, a communally-developed open-source XML database with a desktop GUI interface.[10] Most importantly, BaseX is platform independent and can be installed on any recent Windows or Mac desktop machine like any typical program. The program is also BSD-licensed, meaning it is free to download and use with attribution. The best thing about BaseX GUI is its simplicity. By default, the interface has four major panes, or windows: an Editor pane, which includes a Project pane for selecting files; a Query Info pane; and a Results pane. The Editor and Results panes are most important to new users that might be overwhelmed by seeing too much of the back-end. The Editor pane functions much like a basic text editor: XQuery scripts can be written manually and files can be opened or saved using icons. After a script is written, a user can press the “Execute query” icon, which looks like a play symbol, and the result shows up in the Results pane. Here the result can be saved or overwritten by a new result by running another query. This simplicity allows for easy trial and error, important for archivists inexperienced with programming to experiment with manipulating data.

[9] Zorba: The NoSQL Query Processor: www.zorba.io.
[10] BaseX: The XML Database: http://www.basex.org.



```
xquery version "3.0";

for $TypeQueryHere in doc('source.xml')
return $TypeQueryHere
```

```
<text>And See Your Results Here!</text>
```

```
Total Time: 330.64 ms

Compiling:
- pre-evaluating fn:doc("source.xml")
- rewriting singleton for to let
- inlining let $TypeQueryHere_0 := document-node { "source.xml" }
- simplifying flwor expression

Query:
xquery version "3.0"; for $TypeQueryHere in doc('source.xml') return $TypeQueryHere

Optimized Query:
document-node { "source.xml" }

Result:
- Hit(s): 1 Item
- Updated: 0 Items
- Printed: 39 Bytes
- Read Locking: global
- Write Locking: none

Timing:
- Parsing: 0.45 ms
```

Figure 1: Screenshot of BaseX GUI running on Windows 7

Learning the Language, Structure, and Syntax

There are two sections of an XQuery script: a *prolog* and a *body*. The prolog is optional, comes before the body, and acts like a header to declare namespaces, variables and some structural parts of XML results such as encoding. Since the goal of this piece is to get archivists started with XQuery as quickly and easily as possible, we will only declare a version and an input document in our prolog:

```
xquery version "3.0";
declare variable $doc := doc(path/to/EAD.xml);
```

The first line specifies the version of the XQuery script and the second line declares the variable `$doc` as the EAD file we want to query. Variables can be any single word with no spaces preceded by a dollar sign ($). We could just as easily use `$ead`, `$xml`, or whatever we like in place of `$doc`. The next `doc()` in the second line is not a variable, but a function that specifically reads an xml document. Users can also replace the `doc()` function with `collection()` and point to a folder instead of a file. This will assign multiple files in a directory to the `$doc` variable. Each statement in the prolog is followed by a semicolon

(;), and after the last prolog statement, the query body begins. Now when we query the variable $doc later in the body, BaseX will know that we mean our EAD file.

In the query body there are two main types of statements you will use most often: FLOWR expressions and IF expressions. FLOWR stands for For, Let, Order by, Where, and Return – the likely questions you will ask of your data (essentially the who, what, where, why, and when). Here is a sample FLOWR expression for querying an EAD file:

```
xquery version "3.0";
declare variable $doc := doc(path/to/EAD.xml);

for $series in $doc/ead/archdesc/dsc/c01
let $extent := $series/did/physdesc/extent
where $series/@level = 'series'
order by  $extent
return $series/did/unittitle
```

For is the first part of a FLOWR expression, think of it as "for each," as the rest of the expression will be performed on each tag it finds in the path given. This statement looks for each <c01> tag in the container list within the variable $doc (our EAD file) and assigns each tag it discovers to the variable $series. We just selected each top-level series from our EAD file. The second line declares another variable ($extent) to represent the contents of each series' <extent> tag within its <did> and <physdesc> tags. The third line **where** statement limits all <c01> tags to only those which a "level" attribute with the value "series" (<c01 level="series"/>). The fourth line orders each remaining <c01> tag by the variable ($extent) declared in line two. In our department, the <extent> tag contains each series' extent in cubic feet, so this line re-orders each <c01> tag by its extent. Finally, the fourth line returns a result. We could ask to just **return** $series, but that would return each <c01> tag as well its contents – including its <did> tag and any lower components in the container list. Thus we will only ask specifically for each <c01> tag's title. The result of this query will return each series title ordered by size. Archivists looking to experiment with XQuery should copy and paste this code into BaseX's Editor pane and point the path in the prolog to an EAD file. Next you should experiment by erasing some code and adding to it in stages to see how each change alters the results. FLOWR expressions only require one of either **`for`** or **`let`** and a **`return`**.

Now we can also add our own XML tags to the result, differentiating from the query return with curly brackets ({ and }). This adds greatly to the utility of XQuery but also introduces some additional complexity.

```
xquery version "3.0";
declare variable $doc := doc(path/to/EAD.xml);

for $series in $doc/ead/archdesc/dsc/c01
let $extent := $series/did/physdesc
where $series/@level = 'series'
order by  $extent
return <FileSeries>{$series/did/unittitle}</FileSeries>
```

If we plug this query into BaseX we will realize that this expression results in each series also retaining its original `<unittitle>` tag. Using the `data()` function will only return the contents of the selected tag. Replace the last line in the code above with the line here:

```
return <FileSeries>{data($series/did/unittitle)}</FileSeries>
```

Functions are the magic words that make XQuery so powerful. They perform actions on the contents within their parenthesis. So far we have used two functions: `data()` and `doc()` in our prolog. The XQuery 3.0 standard includes 225 built-in functions and many more are available on the FunctX XQuery Function Library website.[11] Often, the simplest place to enter these functions is within a **`let`** clause. Here we have our same query with the addition of the `upper-case()` function.

```
xquery version "3.0";
declare variable $doc := doc(path/to/EAD.xml);

for $series in $doc/ead/archdesc/dsc/c01
let $extent := $series/did/physdesc
where $series/@level = 'series'
let $UPPERseries := upper-case($series/did/unittitle)
order by  $extent
return <FileSeries>{$UPPERseries}</FileSeries>
```

[11] FunctX XQuery Functions: http://www.xqueryfunctions.com.

This query declares a new variable `$UPPERseries` which performs the function `upper-case()` on the title of our old `$series` variable. When we return the new variable it will be in all capital letters.

In addition to FLOWR expressions, XQuery users will often employ IF expressions. These expressions are useful to make conditionals that accept the large amount of variability that is often present in archival data.

```
xquery version "3.0";
declare variable $doc := doc(path/to/EAD.xml);

if ($doc/ead/archdesc/did/physdesc/extent < 20)
then (<size>{data($doc/ead/archdesc/did/physdesc/extent)}</size>)
else (<size>This collection is large.</size>)
```

In the first line this IF expression in the XQuery body tests whether the collection is smaller than 20 cubic feet. The second and third lines offer two options that depend on whether or not the first line is true. The query either returns the extent of the collection or text that states the collection is large. You may get an error if the EAD file you are using contains more than a number in the `<extent>` tag. Try experimenting by changing the less than operator (<) to a greater than (>), equal (=), or not equal (!=) operator. A more complex IF expression might draw different results if an EAD container list has more than one or two levels of arrangement.

Manipulating Data from EAD Finding Aids

Now that we have experimented using FLOWR expressions, functions, and IF expressions, we can manipulate our data in many powerful ways. Not only can XML be entered in a **`return`** statement, but an entire query can be placed within XML tags. Here is a more complex XQuery script that summarizes a set of EAD files:

```
xquery version "3.0";

<Repository>
{
  for $collection in collection('path/to/EAD/files')
  let $info := $collection/ead/archdesc
  let $date := $info/did/unitdate[@type = 'inclusive']
  order by $date/@normal
  return
    <Collection>
      <Title>{data($info/did/unittitle)}</Title>
      <Date>{data($date)}</Date>
      <Size>{data($info/did/physdesc)}</Size>
      <Arrangement>
        {
          if ($info/dsc/c01/@level = 'series')
          then
            (
              for $series in $collection/ead/archdesc/dsc/c01
              where $series/@level = 'series'
              return <Series>{data($series/did/unittitle)}</Series>
            )
          else (<Series>This collection contains no series.</Series>)
        }
      </Arrangement>
    </Collection>
}
</Repository>
```

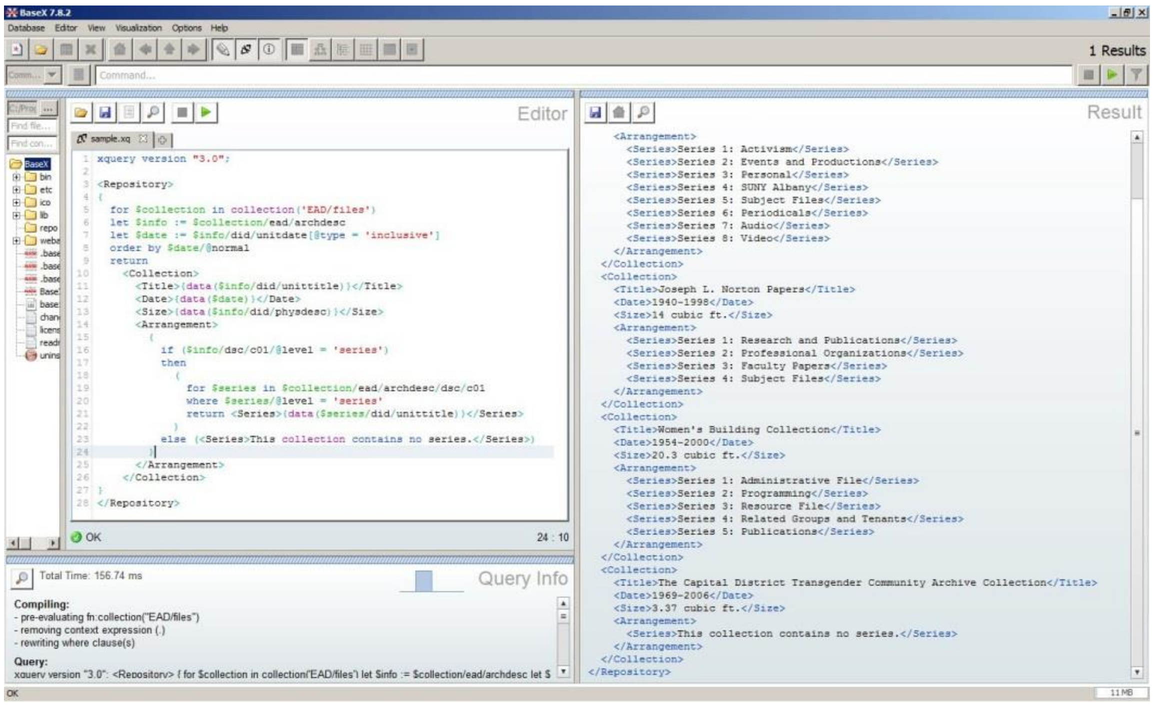
This script produces a `<Repository>` root tag containing a FLOWR expression. The **`for`** statement selects each file in the collection directory `path/to/EAD/files` and assigns it to a variable `$collection`. The first **`let`** statement declares the variable `$info` as a shortcut to the `<archdesc>` tag node. The second **`let`** statement assigns the collection level `<unitdate>` using the shortcut we just stated.

Notice that this statement also uses an XPath predicate to select only the unitdate tags with the attribute `@type="inclusive."` While this condition may also be written as an XQuery IF expression, it is much simpler here to use XPath because it uses only one line. While XPath predicates are necessary in XSLT to perform simple actions like this, in XQuery they are optional but often make queries much simpler.

Since some collections may also have bulk dates, limiting the `$date` variable to just one value allows the script to sort the collections using the **`order by`** statement. This line orders the collections according to their normalized date contained within the `@normal` attribute within the `<unitdate>` tag node.

The return statement adds some additional complexity. Here, for each collection the script produces a `<Collection>` tag with the children `<Title>`, `<Date>`, `<Size>`, and <`Arrangement>`. The first three children contain curly brackets with more XQuery code that produces the contents of the `<unittitle>` tag, the `$date` variable, and the `<physdesc>` tags respectively.

In the `<Arrangement>` tag, the script attempts to list the series titles from each collection. Since one of the collections contains no series, an IF expression is needed. First, the IF expression tests if the collection has any `<c01>` tags that have a `@level` attribute that equals "series." If that statement is true, the Then statement activates another FLOWR expression that selects the `<unittitle>` of each `<c01>` tag and produces it within a `<Series>` tag. Here, there **`where`** statement is not necessary because of the **`if`** statement, but was retained because it shows that users can place a complete FLOWR expression within an IF expression. Lastly, if the If statement is false, the **`else`** statement produces a single `<Series>` tag that states that the collection has no series. The result of this script will be a short summary of collections encoded in EAD (Figure 2: Screenshot of XQuery script and result in BaseX GUI).



```
xquery version "3.0";

<Repository>
{
  for $collection in collection('EAD/files')
  let $info := $collection/ead/archdesc
  let $date := $info/did/unitdate[@type = 'inclusive']
  order by $date/@normal
  return
    <Collection>
      <Title>{data($info/did/unittitle)}</Title>
      <Date>{data($date)}</Date>
      <Size>{data($info/did/physdesc)}</Size>
      <Arrangement>
        {
          if ($info/dsc/c01/@level = 'series')
          then
            (
              for $series in $collection/ead/archdesc/dsc/c01
              where $series/@level = 'series'
              return <Series>{data($series/did/unittitle)}</Series>
            )
          else (<Series>This collection contains no series.</Series>)
        }
      </Arrangement>
    </Collection>
}
</Repository>
```

```
    <Arrangement>
      <Series>Series 1: Activism</Series>
      <Series>Series 2: Events and Productions</Series>
      <Series>Series 3: Personal</Series>
      <Series>Series 4: SUNY Albany</Series>
      <Series>Series 5: Subject Files</Series>
      <Series>Series 6: Periodicals</Series>
      <Series>Series 7: Audio</Series>
      <Series>Series 8: Video</Series>
    </Arrangement>
  </Collection>
  <Collection>
    <Title>Joseph L. Norton Papers</Title>
    <Date>1940-1998</Date>
    <Size>14 cubic ft.</Size>
    <Arrangement>
      <Series>Series 1: Research and Publications</Series>
      <Series>Series 2: Professional Organizations</Series>
      <Series>Series 3: Faculty Papers</Series>
      <Series>Series 4: Subject Files</Series>
    </Arrangement>
  </Collection>
  <Collection>
    <Title>Women's Building Collection</Title>
    <Date>1954-2000</Date>
    <Size>20.3 cubic ft.</Size>
    <Arrangement>
      <Series>Series 1: Administrative File</Series>
      <Series>Series 2: Programming</Series>
      <Series>Series 3: Resource File</Series>
      <Series>Series 4: Related Groups and Tenants</Series>
      <Series>Series 5: Publications</Series>
    </Arrangement>
  </Collection>
  <Collection>
    <Title>The Capital District Transgender Community Archive Collection</Title>
    <Date>1969-2006</Date>
    <Size>3.37 cubic ft.</Size>
    <Arrangement>
      <Series>This collection contains no series.</Series>
    </Arrangement>
  </Collection>
</Repository>
```

```
Total Time: 156.74 ms

Compiling:
- pre-evaluating fn:collection("EAD/files")
- removing context expression (.)
- rewriting where clause(s)

Query:
xquery version "3.0": <Repository> { for $collection in collection('EAD/files') let $info := $collection/ead/archdesc let $
```

Figure 2: Screenshot of XQuery script and result in BaseX GUI

This XQuery script is fairly complex for archivists without prior knowledge of XQuery or any other techniques for querying encoded data. It serves to demonstrate a small portion of the capabilities of XQuery and to give new users another example to experiment with. More complex scripts will force archivists to tackle the XQuery structure, which will likely be unfamiliar. It is very common to experience errors when curly brackets or parenthesis are omitted or misplaced. The best process is to start simple and build up from there. One of the major benefits to BaseX is that new users can run a simple query to confirm it works correctly and add to it later. Then, if errors are produced, users can just undo their changes and go back to a safe starting point.

Results

Archivists with a basic grasp of XQuery will be able to run simple reports and draw select information from their EAD finding aids. More advanced users will be able to reformat large volumes of legacy data, merge and split files, identify errors and inconsistencies, and perhaps derive data from finding aids for outreach, exhibitions, or other purposes.

One of the major hurdles for new XQuery users is to thoroughly understand their data. The structure of XML files can be very complex for users with more experience querying relational databases. Yet, the archives community has devoted substantial time and resources to thoroughly understanding EAD. This gives archivists familiar with EAD a distinct advantage. With a complete understanding of EAD and its structure, archivists with only a basic understanding of XQuery syntax can do tremendous things with their collection data.

There are many possible applications for XQuery. One example is a workflow tool developed at the University at Albany, SUNY to automate the creation of EAD finding aids. This tool utilizes a custom XML schema mapped to a Microsoft Excel workbook. Users enter collection-level data on the first sheet and the container list and series-level description on subsequent sheets. The workbook supports complex arrangements of up to three levels. Next the data is exported as XML and an XQuery script is activated that produces a complete and valid EAD finding aid.

Another possible application example is a tool to acquire metadata for digital images from volunteers or experts over the Internet. An archivist at the Marist Archives and Special Collections developed a prototype for this. The Metadata Creation Tool was designed to run on a web server running the open source eXist-db XML database. An archivist would upload digitized access images as JPEGs with unique IDs in each filename. An XQuery application provided a webpage through which off-site volunteers could select images and enter descriptive information with a simple web form. The web page was simple enough for use with only basic Internet experience. The descriptions were labeled with the unique IDs and entered into a running XML file, which other volunteers could add to simultaneously. An additional interface would be developed for reviewing the descriptions and for quality assurance. The simplicity of XQuery enabled this complex application to be developed in under two weeks, requiring only around 600 lines of code.

The use of XQuery also has one important added benefit: it forces archivists to see encoded finding aids as *data*. The archives community has invested heavily in developing EAD and other XML standards and tools. This is consistent with a long tradition of archivists focusing primarily on storage (both analog and digital) over use. As archivists have discovered in the paper world, it is difficult to have the foresight to develop effective storage processes without understanding how the stored materials will be used. Through learning and making use of XQuery to process EAD finding aids and other XML data, archivists will better understand how that data may be used, and in turn, better understand how to store and manage it.

For a practical example, say a novice XQuery user attempts to run a simple report to sort a list of EAD-encoded collections by extent. If the collection-level extent is encoded as `<extent label="Extent">20.3 cubic ft.</extent>`, the query will fail. By making use of XQuery, and thinking of the `<extent>` tag as a discrete unit of data, it will become clear that, to sort collections by extent, it will need to be encoded as `<extent label="Extent" unit="cubic ft.">20.3</extent>`. In other words, actually using EAD data will help archivists to better understand it.

Further Resources

BaseX: The XML Database, accessed July 9, 2014, http://www.basex.org.

eXist-db: The Open Source Native XML Database, accessed July 9, 2014, http://exist-db.org/exist/apps/homepage/index.html.

eXist-db: Documentation, accessed July 9, 2014, http://exist-db.org/exist/apps/doc.

Fawcett, Joe, Liam R. E. Quin & Danny Ayers, *Beginning XML* (Indianapolis: John Wiley & Sons, Inc., 2012).

FunctX XQuery Functions: Hundreds of Useful Examples, accessed July 9, 2014, http://www.xqueryfunctions.com.

Saxon: The XSLT and XQuery Processor, accessed July 9, 2014, http://saxon.sourceforge.net.

Stack Overflow, accessed July 9, 2014, http://www.stackoverflow.com.

Walmsley, Priscilla, *XQuery* (Sebastopol, CA: O'Reilly Media, Inc., 2007).

XQuery 3.0: An XML Query Language, accessed July 9, 2014, http://www.w3.org/TR/xquery-30.

W3Schools XQuery Tutorial, accessed July 9, 2014, http://www.w3schools.com/xQuery/default.asp.

Zorba: The NoSQL Query Processor, accessed August 6, 2014, http:// www.zorba.io.

Appendix

The following compares a section of an XSLT stylesheet that was part of the original EAD cookbook, with its equivalent in XQuery:

XSLT

```
<xsl:if test="child::unitdate[@type='inclusive']">
    <xsl:choose>
        <xsl:when test="unitdate[@type='inclusive']/@label">
            <tr>
                <td valign="top">
                    <b>
                    <xsl:value-of
select="unitdate[@type='inclusive']/@label"/>
                    </b>
                </td>
                <td>
                    <xsl:value-of select="unitdate[@type='inclusive']"/>
                </td>
            </tr>
        </xsl:when>
        <xsl:otherwise>
            <tr>
                <td valign="top">
                    <b>
                        <xsl:text>Dates: </xsl:text>
                    </b>
                </td>
                <td>
                    <xsl: value-of select="unitdate[@type='inclusive']"/>
                </td>
            </tr>
        </xsl:otherwise>
      </xsl:choose>
</xsl:if>
```

XQuery

```
<tr>{
    for $date in unittitle/unitdate
    where $date/@type = 'inclusive'
    return
        if (exists($date/@label))
        then (<td valign="top">
                <b>{data{$date/@label}</b>
              </td>
              <td>{data($date)}</td>)
        else (<td valign="top">
                <b>Dates:</b>
              </td>
```

```
                  <td>{data($date)}</td>)
}</tr>
```