BaseX for Newbies

Tamara Marnell, Orbis Cascade Alliance
March 2022

Introduction

The documentation of the BaseX project is excellent and extensive. However, if you're new to the world of XML databases, excellent and extensive documentation can be overwhelming.

Without foundational knowledge and a familiarity with the structure and vocabulary of BaseX, even the Getting Started page and BaseX for Dummies documents can make your head spin. What is a "module?" Or "client/server architecture?" Why are there 14 different scripts to start and stop BaseX, and which one am I supposed to use? How come the documentation sometimes looks like "CREATE DB database" and other times like "db:create('database')"? I know "indexing" is a thing I'm supposed to do, but what *is* it, and how can I do it?

This document will provide a conceptual framework for understanding BaseX and what you can use it to accomplish. The examples will use the 71 EAD files from the United States Library of Congress's Prints and Photographs Collection, which are in the public domain and available for worldwide use and reuse under CC0 1.0 Universal. A ZIP copy of the files is available on Google Drive.

What is BaseX?

“BaseX is a light-weight, high-performance and scalable XML Database and an XQuery 3.1 Processor with full support for the W3C Update and Full-Text extensions.”

In newbie terms, BaseX is a Java program you can use to build searchable databases of XML files. **XQuery** is the functional programming language you can use to read, update, or search your databases. XQuery functions use **XPath** expressions to select nodes within a document.

What is client/server architecture?

When you visit a restaurant as a **client**, you place an order with your **server**, and they deliver the food you requested.

Similarly, a “client/server architecture” means one program or machine makes requests for information, and a second program or machine delivers that information. For example, when you visit a website, your browser is the client requesting a page from the server, and the machine hosting the website is the server that sends back the page as HTML.

MySQL is another database program that uses this architecture. If you want to install a WordPress website on a web server, you first need to install and configure the MySQL database server package, and then start the service. While the service is running, you can use the MySQL command-line client to log in and tell the service to create databases, query them, update them, delete them, and so on.

BaseX is like MySQL. The server is the system-wide service that stores the databases you create. The client allows you to interact with the databases by giving the server commands like:

```
SET FTINDEX true
SET STRIPNS true
CREATE DB eads C:\Users\[user]\Documents\loc
CLOSE
```

This will create a database named “eads” with all of the documents in the “loc” folder saved in the user’s Documents, indexed for full-text searching and with namespaces stripped.

```
OPEN eads
ADD C:\Users\[user]\Documents\loc\test.xml
OPTIMIZE
CLOSE
```

This will open the eads database, add the theoretical document “test.xml” to it, rebuild the database indexes, and close the database so other processes can access it. You can test this by creating a new file within “loc” with a simple XML structure, like <test>123</test>, and then running these commands.

```
LIST eads
```
This will produce a list of all resources currently in your “eads” database.

```
Input Path       Type  Content-Type      Size
--------------------------------------------
pp002001.xml     xml   application/xml  1660
pp002002.xml     xml   application/xml  1545
pp002003.xml     xml   application/xml  949
[...]
test.xml         xml   application/xml  3

72 Resource(s).
```

```
XQUERY ft:search('eads', 'film stills')
```
This uses the XQuery function ft:search() to return a sequence of the 14 text nodes in the “eads” database that contain the phrase “film stills.” This function is part of the Full-Text Module (see “What are modules?” below).

```
DROP DB eads
```
This will delete the database named “eads.”

Additionally, you can start other services like REST to interact with the databases in an internet browser. The GUI for Windows is another kind of client that allows you to try creating databases, adding resources, etc. through menus and buttons instead of direct commands, with panels that show you which commands were run and what the results were.

What are modules?

The “modules” referred to in BaseX documentation are XQuery modules: files of XQuery functions with a namespace prefix that you can use to call those functions in other scripts. You can also define your own module of custom functions and import it into your scripts.

For example, the Database Module contains XQuery functions to read, update, and get information about your databases. You’ll see overlap between the functions listed on the page and the commands used as examples above:

```
db:create('eads', 'C:\Users\[user]\Documents\loc', (),
   map{'ftindex':true(),'stripns':true()})
```
Like the command-line example above, this function in XQuery creates a database named “eads” containing the Library of Congress documents as resources, with a full-text index and namespaces stripped.

`db:add('eads', 'C:\Users\[user]\Documents\loc\test.xml'), db:optimize`
These two functions first update the database with the document “test.xml,” and then optimize the database’s full-text and other indexes. The comma separates these into two separate FLWOR expressions returned one after the other (see “What is FLWOR?” below).

`db:drop('eads')`
This deletes the “eads” database.

As you can see, there are different ways to accomplish tasks in BaseX, which is why you’ll sometimes see examples written like “GRANT write TO myuser” (under User Management commands) and sometimes “user:grant(‘myuser’, ‘write’)” (under the User Module). Which one you use will depend on the structure and needs of your project.

What is FLWOR?

FLWOR is the structure of an XQuery expression and stands for For, Let, Where, Order By, and Return. For example:

```
<results>{
for $ead in db:open('eads')/ead
  let $title :=
     normalize-space(string-join($ead/archdesc/did/unittitle/node(),
     ', '))
  let $date := $ead/archdesc/did/unitdate[1]/@normal
  let $start :=
     if (not(contains($date, '/'))) then xs:integer($date)
     else xs:integer(substring-before($date, '/'))
  where $start < 1800
    order by $title ascending
    return <result>{$title}</result>
}</results>
```

In this expression:

1. The “for” clause goes through the root <ead> node of each resource in the “eads” database
2. The “let” clauses assign values from the resources to variables:
    a. $title - the nodes within /ead/archdesc/did/unittitle concatenated by commas
    b. $date - the “normal” attribute of the first element within /ead/archdesc/did/unitdate
    c. $start - an integer representing the first year in the date, which is either the first half of an inclusive range like “1400/1900” or the full year like “1984”
3. The “where” clause restricts the records to only those with starting dates before 1800
4. The “order by” clause orders the records alphabetically by title
5. The “return” clause prints out the <result> node

The results below represent all finding aids in this collection with starting dates before 1800 CE.

```
<results>
  <result>Alfred Bendiner Memorial Collection (Library of Congress),
     1706-1968, (bulk 1943-1964)</result>
  <result>Bulfinch architectural drawing archive (Library of
     Congress), 1787-1860, 1787-1832</result>
  <result>Latrobe architectural drawing archive (Library of
     Congress), 1795-1850, 1795-1820</result>
  <result>Ruthven Deane bookplate collection, 1600-1934</result>
  <result>Study collection of 18th and 19th-century American book
     illustrations and portrait prints, ca. 1770 - ca. 1850</result>
  <result>Wilhelm Schreiber Collection of 15th-18th century book
     illustrations, 1400-1900, 1490-1690</result>
</results>
```

FLWOR expressions can also be nested within “let” and “return” clauses or XQuery functions. You can see a nested expression in the calculation of $start above–the if/then statement is actually a return clause. The word “return” is needed for a return clause only if you used other clauses before it.

Here’s a more complex example with multiple nested FLWOR expressions, which returns genre/form headings that contain the word “print” and the files they appear in.

```
let $map := map:merge(
  for $ead in db:open('eads')/ead
    let $file := db:path($ead)
    let $genreforms := $ead//controlaccess/genreform
    for $genreform in $genreforms
      let $genreform_text := substring-before($genreform, '--')
      where contains(lower-case($genreform_text), 'print')
        return map:entry($genreform_text, $file),
  map{"duplicates":"combine"}
)
return <genreforms>{
for $genreform in map:keys($map)
  let $files := $map($genreform)
  let $count := count($files)
  order by $count descending
    return <genreform text="{$genreform}" count="{$count}">{
      for $file in $files
        return <file>{$file}</file>
    }</genreform>
}</genreforms>
```

In this example, we first use a FLWOR expression within map:merge() to get all genre/form headings in the EADs and their files. The return of this expression is another nested FLWOR expression starting “for $genreform in $genreforms,” which returns map entries of genre/forms containing “print” and their files.

Then we return the results we want to see in <genreforms>, ordered by the count of files. A final nested FLWOR expression goes through the sequence $files and prints out each as a <file>.

```
<genreforms>
  <genreform text="Gelatin silver prints" count="21">
     <file>pp002005.xml</file>
     <file>pp002008.xml</file>
     <file>pp002010.xml</file>
     […]
  </genreform>
  <genreform text="Photographic prints" count="15">
     <file>pp002009.xml</file>
     <file>pp002011.xml</file>
     <file>pp020001.xml</file>
     […]
  </genreform>
  <genreform text="Photomechanical prints" count="9">
     <file>pp002010.xml</file>
     <file>pp002011.xml</file>
     <file>pp020006.xml</file>
     […]
  </genreform>
  […]
</genreforms>
```

Another clause not in the FLWOR acronym is “group by,” which you can use to group multiple nodes under unique values.

The following will perform a full-text search of the “eads” database for the word “portrait” and return the count of results grouped by the titles of their parent records. Note that “hits” represents the number of text nodes within that EAD that contain the word “portrait,” which could be less than the total instances of the word “portrait” if it appears more than once in a text node.

```
<results>{
for $result in ft:search('eads', 'portrait')
  let $ead := $result/ancestor::ead
  let $title :=
    normalize-space(string-join($ead/archdesc/did/
    unittitle/node(), ', '))
  group by $title
    let $hits := count($result)
    order by $hits descending
    return <result>
      <title>{$title}</title>
      <hits>{$hits}</hits>
    </result>
}</results>
```

Results:

```
<results>
  <result>
     <title>Ruthven Deane bookplate collection, 1600-1934</title>
     <hits>342</hits>
  </result>
  <result>
     <title>New York World-Telegram and the Sun Newspaper Photograph
     Collection, Biographical File, A to L, ca. 1880-1967,
     1920-1967</title>
     <hits>268</hits>
  </result>
  <result>
     <title>Visual materials from the Walt Whitman papers in the
     Charles E. Feinberg collection, 1848-1969</title>
     <hits>240</hits>
  </result>
  [...]
</results>
```

Which script should I use to start BaseX?

If you download the ZIP distribution of the latest BaseX version, in the “bin” folder you’ll see fourteen different scripts. Which one you use to start BaseX depends on what you want to do with your project. All require Java 8 or greater to be installed and running on your machine.

For testing on a Windows machine, you can double-click the BaseX.jar file in the root of the folder, and everything you need to get started will be up and running.

For building a website with BaseX as a service, run either *basexserver* (for the database server alone) or *basexhttp* (for both the database server and the HTTP server for REST) as a background process. On our Ubuntu 20.04 AWS instance, where I have BaseX installed under /opt/basex, I placed the following in /etc/rc.local to start the database server every time the operating system boots:

```
#!/bin/sh
/opt/basex/bin/basexserver &
```

To shut down the service in Linux–to install a version update or make configuration changes, for example–run *basexserverstop* or *basexhttpstop*.

If you’re building an application or website, you can interact with the BaseX database server through the Client for your language on port 1984 (by default). For example, in PHP, after installing and requiring the client scripts you can build and query our test EADs database like this:

```
use BaseXClient\BaseXException;
use BaseXClient\Session;

try {
  // Start the session
  $session = new Session("localhost", 1984, BASEX_USER,
   BASEX_PASSWORD);

  // Build the ead database with documents in loc
  $session->execute('SET FTINDEX true');
  $session->execute('SET STRIPNS true');
  $session->execute('CREATE DB eads /home/[user]/public_html/loc');
  $session->execute('CLOSE');

  // Query the database
  $input =
   file_get_contents('/home/[user]/public_html/xquery/get-urls.xq');
  $query = $session->query($input);
  $query->bind('db', 'eads');
  $results = $query->execute();
  $query->close();

  // Close the session
  $session->close();
}
catch (BaseXException $e) {
  print $e->getMessage();
}
```

In this example, the XQuery file *get-urls.xq* takes an external variable $db set to the name “eads”, and the PHP variable $results will contain the output.

The XQuery file contents might look like:

```
declare variable $db as xs:string external;
<urls>{
for $ead in db:open($db)/ead
  let $url := string($ead/control/recordid/@instanceurl)
  let $node_id := db:node-id($ead)
  return <url node="{$node_id}">{$url}</url>
}</urls>
```

Combined, these scripts would return an XML document with the Library of Congress URLs of all EADs in the “eads” database, with the BaseX node IDs of the root elements as “node” attributes. (See Node Storage.)

```
<urls>
  <url node="4">https://hdl.loc.gov/loc.pnp/eadpnp.pp002001</url>
  <url node="1664">https://hdl.loc.gov/loc.pnp/eadpnp.pp002002</url>
  <url node="3209">https://hdl.loc.gov/loc.pnp/eadpnp.pp002003</url>
  [...]
</urls>
```

The REST service is another way to run XQuery files. If you run *basexhttp* to start both the database server and HTTP server, you can access the REST service through port 8984 (by default). Then you can get the results of a query with JavaScript, for example, by sending a POST or GET request to http://yourdomain.com:8984/rest?run=get-urls.xq&db=eads

If you plan to use a client library for a server-side language, running the REST service in addition to the database server is not necessary, because you can write your own scripts to deliver asynchronous BaseX results from port 1984. However, in my development experiments I didn’t see a performance difference between using the REST service or the client, so you can run your queries whichever way you prefer. BaseX for Dummies demonstrates how to get results from the REST server with PHP starting on page 9.

What am I supposed to put where?

When you install BaseX, you can configure where you want to store certain folders and files in the .basex configuration file.

- DBPATH - Where BaseX will write the database, user, and log files.
- REPOPATH - Where BaseX will look for and load any XQuery modules you’ve installed or written yourself.
- WEBPATH - The directory of web application contents, if you’re using those services.
- RESTPATH - Where BaseX will look for the XQuery files called in the “run” parameter of a REST request, if you’re running that service.
- RESTXQPATH - Where BaseX will look for RESTXQ modules.

When you create a database, you can point at a collection of files anywhere on your machine, as demonstrated in the command and XQuery examples above. Note that after you ingest the documents into a database, the "path" in BaseX to that resource will become [database]/[filename], which is most likely different from the physical path to the file.

Physical path:	C:\Users\[user]\Documents\loc\pp021024.xml
Input path:	pp021024.xml
Resource in XQuery:	doc('eads/pp021020.xml')

If you want to use the File Module, paths are relative to the working directory, or wherever you triggered the script used to launch the server. If on a Windows machine you opened the BaseX GUI from C:\Users\[user]\Program Files (x86)\BaseX\BaseX.jar, your path for File module functions will need to be the absolute path or "..\..\Documents\loc\[file]".

What is an index, and how do I make one?

In print reference books, the **index** is a section in the back of the book that lists the key topics covered and their relevant page numbers.

For example, a theoretical 350-page textbook about BaseX might have an index that looks like this:

```
Clients…………………………………10-25
Commands……………………………54-65, 210-218
XQuery Modules…………………7, 49, 300-329
```

If you want to read about BaseX commands, you don't have to go through all 350 pages to find them. You can look in the back for the word "commands" and turn straight to the sections starting page 54 and page 210 to get the information you want.

Database indexes do the same thing: isolate key terms with direct pointers so you can get the information you need quickly, instead of waiting while the program sifts through a lot of irrelevant data.

Some BaseX Indexes will be built automatically, like the indexes of node names and attributes. Some can be optionally turned on for databases that need them, like the full text index. If you have specific nodes or attributes you'll want to query quickly and/or frequently, you can build a database of your own with that information.

For example, if you have many EAD documents in a database and will want to fetch the ones that have specific text in a control access <subject> node, you can create a custom "subject-index" database out of the "eads" database:

```
let $index := <subjects>{
for $subjects in db:open('eads')//controlaccess/subject
  let $term := normalize-space($subjects)
  group by $term
  where not ($term="")
    return <subject term="{$term}">{
      for $id in
        distinct-values($subjects/ancestor::ead/control/recordid)
        return <id>{$id}</id>
    }</subject>
}</subjects>
return db:create('subject-index', $index, 'subjects.xml')
```

The result saved in subject-index would look like:

```
<subjects>
  <subject term="African Americans--1830-1880.">
     <id>loc.pnp/eadpnp.pp002001</id>
  </subject>
  <subject term="Animals--1830-1880.">
     <id>loc.pnp/eadpnp.pp002001</id>
  </subject>
  <subject term="Children--United States--1830-1880.">
     <id>loc.pnp/eadpnp.pp002001</id>
  </subject>
  [...]
</subjects>
```

Then a query for db:open('subject-index')/subjects/subject[@term="Artists--1920-1970."], for example, would return the four finding aid record IDs with that exact subject, without searching through all documents again.

In another use case, the following would return a list of all subjects associated with a subset of documents passed to the query as a string of record IDs separated by commas:

```
declare variable $i as xs:string external;
<terms>{
let $ids := tokenize($i, ',')
for $subject in db:open('subject-index')/subjects/subject[id=$ids]
  let $count := count($subject/id[text()=$ids])
  order by $count descending
  return <term count="{$count}">{$subject/@term}</term>
}</terms>
```

For a small database like our sample Library of Congress finding aids, you won't see much of a difference in speed between querying "eads" and querying "subject-index." However, in a real-world environment a repository might have 50,000 documents spread across 50 databases, and querying one index will be much faster than opening 50 databases and navigating through each XML tree for the //controlaccess/subject nodes.

To keep indexes up to date, you can turn on the UPINDEX option to update the built-in indexes every time a database is updated, which can be slow; or you can optimize your databases after batch update operations. For example, this XQuery adds new documents to the "eads" database and then optimizes it to update the indexes. It takes a string of filenames separated by commas as the variable $files.

```
declare variable $files as xs:string external;
let $add_files := %updating function($files) {
  for $file in tokenize($files, ',')
    return db:add('eads', $file)
}
return updating $add_files($files), db:optimize('eads')
```

To keep a custom index updated, you need XQuery scripts to keep up with additions, deletions, and replacements as well. The following indexes the subjects of the same string of files that were added above to "eads":

```
declare variable $files as xs:string external;
let $add_to_subjects := %updating function($files) {
  for $file in tokenize($files, ',')
    let $doc := doc('eads/' || $file)
    let $id := $doc/ead/control/recordid
    for $term in $doc//controlaccess/subject
      let $existing_subject :=
        db:open('subject-index')/subjects/subject[@term=$term]
      where not ($existing_subject/id[text()=$id])
        return
          if (exists($existing_subject)) then
            insert node <id>{$id}</id> as last into $existing_subject
          else
            insert node <subject term="{$term}"><id>{$id}</id></term>
            as last into db:open('subject-index')/subjects
}
return updating $add_to_subjects($files),
db:optimize('subject-index')
```

How can I make full-text searches faster?

Though I've showed examples of enabling the full-text index on a database of finding aids and using ft:search() to get results, using a database of original documents for public searching will be slow for multiple reasons.

1. The full-text index is at the text node level. If you want to get results at the document level, navigating up from a text result to the root of a document and grouping by an identifier is expensive.
2. BaseX completes one read or write operation on a database at a time. If an editor is attempting to update the same database that a user is attempting to search, one operation will lock out the other. (See Transaction Management.)

If your project includes a full-text search, it will be more efficient to build a dedicated text index of "tokens" that are preprocessed to convert strings to lowercase, remove punctuation, and remove unneeded stopwords that can bloat the index like "the," "a," etc.

```
let $index := <eads>{
  for $ead in db:open('eads')/ead
    let $tokens := ft:tokenize(string-join($ead//text(), ' ')
    let $id := $doc/ead/control/recordid
    return <ead id="{$id}">{$tokens}</ead>
}</eads>
return db:create('text-index', $index,
  map{'ftindex':true(),'stopwords':'/opt/basex/etc/stopwords.txt'})
```

The "stopwords" option above is the path to your file of stop words, separated by line breaks. You can also remove the stopwords from the tokens yourself, so they won't be stored if they're not needed. The function below returns a sequence of tokens that do not match any entries in a sequence of stopwords, which you could pass to the XQuery as an external variable or store in another BaseX database.

```
declare function local:remove_stopwords($tokens as xs:string*,
$stopwords as xs:string+)  {
  string-join($tokens[not(. = $stopwords)], ' ')
};
```

You can also create both "production" and "working" copies of your index databases, so users can search a "production" copy at the same time an editor is updating the "working" copy (CPU and memory allowing). You can accomplish this by utilizing db:copy() after creating or updating an index.

```
db:copy('text-index', 'text-index-prod')
```

How can I make other queries more efficient?

As you craft your databases and XQuery expressions, you'll probably write some that take a long time or even time out. Here are some general tips for improving them.

Avoid ambiguous XPath expressions.

It's tempting to use //name to jump straight to the node you want, but this tells XPath to look for <name> under all descendents. Similarly, "ancestor::name" looks through every parent up the tree. In production applications, use direct paths whenever possible, and ambiguous paths like "//" or "ancestor" only if the location of the target node is unknown.

Other XPath expressions that can be very slow include trying to find a node with //*[local-name="name"] or similar, to work around namespaces.

Strip namespaces.

In the Library of Congress database examples, I stripped the namespaces with the option STRIPNS. If I didn't, in every XQuery script I would need to declare the EAD3 namespace with a prefix like "ead," and then use the prefix in every XPath expression, like this:

```
declare namespace ead="http://ead3.archivists.org/schema/";
<ids>{
for $id in db:open('eads')/ead:ead/ead:control/ead:recordid
  return <id>{$id}</id>
}</ids>
```

Namespaces can also cause headaches if some of your documents have them while others don't. If you can, delete namespaces from the documents before adding them to BaseX or take advantage of the STRIPNS option.

Check the logs.

What seems like a "slow" query could be a query with an error that's returning no results. You can see error printouts in the data/.logs directory of your BaseX installation and/or in the bottom right window of the Windows GUI.

Try breaking up huge databases into smaller databases, or consolidate tiny databases into bigger ones.

Read operations on one database will be faster than looping through many databases. For example, if you have 50 document database IDs and pass the ones to search as a string separated by commas in the external variable $d, reading one index with the IDs stored as attributes in the entries (Example A) will be faster than reading 50 different indexes with the IDs in the names (Example B).

Example A:

```
let $dbs := tokenize($d, ',')
for $result in ft:search('text-index', $terms)/parent::ead
  where $result/@db = $dbs
    return $result
```

Example B

```
for $d in tokenize($d, ',')
  ft:search('text-index' || $d, $terms)/parent::ead
    return $result
```

However, indexing the text of all 50 document databases in one database will result in slower update operations. Every time an editor adds a new document and text-index needs to be optimized, BaseX could launch a process that maxes out CPU for many seconds or minutes. Even if you have a production copy for searching, all users of your application will encounter long wait times if your machine has no more computing power or memory to spare.

Try building both consolidated databases and split databases with your documents, and see which hits the right balance for your organization. In the case of Archives West, we have combined control access heading indexes, but individual text indexes for each of our document databases, because the 30-second slow-down caused by optimizing a single text index was more noticeable than the 2-second slow-down caused by opening many databases to search.

Perform update operations in batches.

Optimizing even relatively small or medium-sized databases is time-consuming. If you optimize every time you make an addition, deletion, or replacement, you can slow your server down more than necessary.

In the previous example of updating a text index after adding new files, I defined an “updating” function to run through a sequence of files and add them all, and then a second FLWOR expression to optimize the database after all additions:

```
return updating $add_files($files), db:optimize('eads')
```

This allows editors to perform batch uploads of 20 files that will not lock up the server with 20 sequential optimizations. It also allows for scheduling the update operations during off-peak hours, either by setting up a cron job on your server or using the BaseX Jobs Module.

Nothing is working. What am I doing wrong?

Here are some stumbling blocks I ran into when first installing BaseX and developing the [Archives West]() website.

The user running BaseX doesn't have write permission for /data.

The user running BaseX needs read, write, and execute permissions to the /data directory, or to whichever directory you defined in DBPATH. If the permissions are incorrect, any changes you make to BaseX user accounts and databases will not be saved.

For example, you might create a user in the client, and BaseX says the user was successfully created. But when you exit out and try to sign in as that user, BaseX says permission is denied. Check whether a users.xml file exists in the data directory. If not, you must give whichever Windows or Linux user will be running BaseX write access. If you'll be creating and updating databases programmatically, the owner might need to be www-data or similar.

The scripts in /bin aren't in your operating system's path or aren't executable.

If you try to run "basexserver" and your system says that command is not found, you'll need to either use the absolute path to the script (e.g., /opt/basex/bin/basexserver) or add the /bin directory to your path variable. In Windows 11, look for "Edit the system environment variables" in the control panel. In Linux, you'll need to run a script on start-up that modifies $PATH.

Also make sure that the user account trying to run the scripts has execute permissions. If you accidentally set the permissions to read only, BaseX can't be launched.

A script says it's out of memory or interrupted.

First, your script could be inefficient. If you're using a server-side programming language, check that you don't have a "while" loop that never closes, and you're not repeating expensive operations like optimizations more than necessary.

Second, the heap size for BaseX is defined in the start-up scripts. By default, it's either 1200 MB or 2 GB, depending on the script. You can increase this to eliminate some out-of-memory errors.

If you're seeing messages that a script was "interrupted," it probably timed out. The timeout limit is defined in .basex, and by default it's 30 seconds. You can increase this to 60 or more if your database creation or updating operations take longer. Make sure other timeout variables in your system match or exceed this value, like in php.ini.

Queries take too long to complete.

If you’ve taken care of all the inefficiencies you can find, but your queries are still slower than expected or desired, your hardware might need to be upgraded.

Archives West has about 45,000 documents, and I initially put it on an AWS instance with 1 vCPU and 4 GB of memory. Upgrading to an instance with 2 vCPU 8 GB of memory doubled the speeds of full-text searches. Test upgrades to a compute-optimized instance with 4 vCPU increased speeds only for unusually expensive queries, but didn’t improve speeds for the types of searches most of our users would submit; and test upgrades to a memory-optimized instance with 16 GB of memory only reduced the percentage of memory used by Java, but didn’t improve performance.

Utilize your Task Manager in Windows or commands like “top” in Linux to see how your machine performs during queries. If you can, experiment with different server types to find one that matches your needs. AWS offers on-demand instances with hourly rates, so for a few U.S. dollars I could launch different instance types for a couple of hours to run speed trials.

Conclusion

BaseX is a powerful tool for organizations that need to work with large collections of XML files, but it’s not a “plug-and-play” tool. Implementation can have a high learning curve, especially if you’re new to XQuery and XPath.

To get started:

1. Download and install BaseX on your machine.
2. Start the BaseX server and a client, either by running BaseX.jar for the GUI or by running the start-up scripts for the command line.
3. Create a test database with either the Library of Congress files from this tutorial or your own chosen documents.
4. Use the GUI Database menu or LIST and INFO commands to see how your resources are represented in BaseX.
5. Practice submitting XQuery expressions in the GUI or command-line client to get results.
6. Practice creating custom index databases out of your resources.
7. Practice getting XQuery results from either the REST service or the client for your programming language.

Ultimately the performance of BaseX in your application will depend on how you structure your indexes, queries, and jobs. Creating a document database is only the first step, and experimentation is part of the process. The BaseX-Talk listserv and BaseX tag on Stack Overflow are great resources to learn more and troubleshoot.