CIS 455/555: Internet and Web Systems

Fall 2020

Homework 2: Web Crawling and Stream Processing

Milestone 1 due October 28, 2020, at 10:00pm ET
Milestone 2 due November 11, 2020, at 10:00pm ET

1. Background

In this assignment, you will explore several Web technologies and continue to build components useful towards your course project, by building a topic-specific Web crawler. A topic-specific crawler looks for documents or data matching certain phrases or content in the header or body.

This assignment will entail in Milestone 1:

- Expanding a dynamic Web application, which runs on the framework you built for Assignment 1 (or Spark Framework) and allows users to (1) create user identities, (2) topic-specific "channels" defined by a set of XPath expressions, and (3) to display documents that match a channel;
- Implementing and expanding a persistent data store (using Oracle Berkeley DB) to hold retrieved HTML/XML documents and channel definitions.
- Fleshing out a crawler that traverses the Web, looking for HTML and XML documents that match one of the patterns.

In Milestone 2:

- Refactoring the crawler to fit into a stream processing system's basic abstractions;
- Routing documents from the crawler through a stream engine for processing one at a time;
- Writing a pattern matcher that determines if an HTML or XML document matches one of a set of patterns;

The resulting application will be very versatile, able to filter documents into categories with specified keywords. Assignment 2 can build on your application server from Assignment 1. However, if you are not confident that your web server is working well, please use Spark Framework (http://www.sparkjava.com/) to test your application. (Using your own server will earn you +5% extra credit. You are allowed to make fixes to it as necessary.)

2. Developing and running your code

You should **fork**, **clone**, and **import** the framework code for HW2 using the same process as for HW0 and HW1 (fork from ssh://git@bitbucket.org/upenn-cis555/555-hw2.git to your own private repository, then clone from your repository to your VirtualBox/Vagrant instance). You should, of course, regularly commit code as you make changes so you can revert; and you should periodically push to your own repository on bitbucket, in case your computer crashes.

Initially you will be using the Spark Framework for this assignment, but for extra credit you can run it using your own HW1 framework (see Section 6).

Carefully read the entire assignment (both milestones) from front to back and make a list of the features you need to implement. There are many important details that are easily overlooked! Spend at least some time thinking about the *design* of your solution. What classes will you need? How many threads will there be? What will their interfaces look like? Which data structures need synchronization? And so on.

We strongly recommend that you regularly check the discussions on Piazza for clarifications and solutions to common problems.

3. Milestone 1: Crawler Manager, B+ Tree Storage, and Crawler

For the first milestone, your task is to **crawl and store Web documents**. These will ultimately be fed into a pattern matching engine for Milestone 2.

3.1 Routes-Based Web Interface / Crawler Manager

In preparation for Milestone 2, we will have a Web interface for login. Your main class should be called `edu.upenn.cis455.crawler.WebInterface` and it should register routes for various behaviors. We have given you a partial implementation of the login handler so you can get started.

1. If the user makes a request for a page and is not logged in (has no Session), the server should output the login form, **login-form.html**. This form should submit a `POST` message as described below under "log in to existing account."
2. If the user *is logged in* (has a session), requests to the root URL (or /index.html) should present a **simple login page showing "Welcome " followed by the user's username**. (We will add more functionality in Milestone 2.)

You will see in the provided code how **Filters** allow you to make decisions about whether the user's request should proceed, or the user should be redirected to the login form. So some of the above should already be present.

Beyond the above, you should build additional routes to support the following functions:

- **Create a new account**. This should take a POST to URL `localhost:45555/register` with **two parameters**, `username` and `password`. Upon success it should return an appropriate success code with a link to the main page and its login screen. On failure, it should return an appropriate error.
- **Log into an existing account** (multiple users should be able to log in at the same time). This should take a POST to URL `localhost:45555/login` with two parameters, `username` and `password`. Upon success it should create a new **Session** with the user's info and return the user to the main page, which should show the logged-in info as above. The **Session should time out after 5 minutes of inactivity.**
- **Log out** of the account that is currently logged in, via `/logout`. Upon success it should redirect the user to the login page.

Note that you may take advantage of **Sessions** and the various other capabilities of the Spark Framework and/or the ones you developed in Homework 1 Milestone 2. Some of the above functionality is provided, so please look carefully to understand what is and is not there.

3.2. Storage of Document and User Credentials in a B+ Tree

We will use Berkeley DB Java Edition (http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html), which may be downloaded freely from their website, to implement a disk-backed data store. Berkeley DB is a popular embedded database, and it is relatively easy to use as a key-value store; there is ample documentation and reference information available on its Web site, e.g., the Getting Started Guide. Berkeley DB has several different ways of storing and retrieving data; perhaps the simplest is through the Java Collections interface.

Your store will hold (at least):

- the usernames and **encrypted passwords** of registered users (see below),
- (in Milestone 2) information about user channels
- and the raw content of HTML or XML files retrieved from the Web, as well as the time the file was last checked by the crawler.

If you use the Collections interface, you will create objects, representing your data, that extend `java.io.Serializable` and store them in objects like `StoredSortedMap`s. **User passwords should instead be saved using SHA-256 hashing. No cleartext passwords should be saved.**

The WebInterface program, when run from the command-line, should take as the first argument a path for the BerkeleyDB data storage instance, and as a second argument, a path to your static HTML files. You should create a data storage directory if it does not already exist.

3.3. Basic Web Crawler

Your web crawler will initially be a Java application that can be run in Eclipse by creating a Run Configuration (as in HW0) with the goal "`clean install exec:java@crawler`". From the command

line, you can also run `mvn exec:java@crawler`. In both cases, the crawler will take the following command-line arguments (in this specific order, and the first three are required):

1. The URL of the Web page at which to start. Note that there are several ways to open the URL.
    a. For plain HTTP URLs you will probably get the best performance by just opening a socket to the port (we've provided the `URLInfo` class to help parse the pieces out). It is also acceptable to use Java's `HTTPUrlConnection`.
    b. For HTTPS URLs you may want to use `java.net.URL`'s `openConnection()` method and cast to `javax.net.ssl.HttpsURLConnection`. This in turn has input and output streams as usual. Here you can keep relying on the sample code.
2. The **directory containing the BerkeleyDB database environment** that holds your store (this will match the path the `WebInterface` takes). As above, the directory should be created if it does not already exist. Your crawler should recursively follow links from the page it starts on.
3. The **maximum size, in megabytes, of documents** to be retrieved from a Web server.
4. An **optional argument indicating the number of files** (HTML and XML) to retrieve before exiting. This will be useful for testing!

The crawler is intended to be run periodically, either by hand or from an automated tool like the `cron` command. **It is, therefore, not necessary to build a connection from the Web interface to the crawler, except that the two will share a common database. Note also that BerkeleyDB does not like to share database instances across concurrent programs so it's okay to assume only one runs at a time.**

The crawler traverses links in HTML documents. You can extract these using a HTML parser, such as JSoup (https://jsoup.org/, included with your Maven package), or simply by searching the HTML document for occurrences of the pattern `href="URL"` and its subtle variations.

If a link points to another HTML document, it should be retrieved and scanned for links as well. The same is true if it points to an XML or RSS document. **Don't bother crawling images or trying to extract links from XML files.** All retrieved HTML and XML documents should be stored in the BerkeleyDB database (so that the crawler does not have to retrieve them again if they do not change before the next crawl). The crawler **must be careful not to search the same page multiple times during a given crawl**, and it should exit when it has no more pages to crawl. You'll need to understand what parts of functionality are provided and where you need to supplement.

Redundant documents and cyclic crawls: Your crawler should compute an MD5 hash of every document that is fetched, and store this in a "content seen" table in BerkeleyDB. If you crawl a document with a matching hash during the same crawler run, you should not index it or traverse its outgoing links.

When your crawler is processing a new HTML or XML page, it should print a **short status report to the Apache Log4J logger**, using the "**info**" status level. At the very least, you should print: "http://xyz.com/index.html: downloading" (if the page is actually downloaded) or "http://abc.com/def.html: not modified" (if the page is not downloaded because it has not changed). Make sure you follow the above format to comply with the autograder's assumptions.

3.4. Politeness

Your crawler must be a considerate Web citizen. First, it must respect the **robots.txt** file, as described in A Standard for Robot Exclusion (http://www.robotstxt.org/robotstxt.html). It must support the **Crawl-Delay** directive (see http://en.wikipedia.org/wiki/Robots.txt) and "**User-agent**: *", but it need not support wildcards in the Disallow: paths. Second, it must always send a HEAD request first to determine the type and size of a file on a Web server. If the file has the type `text/html` or one of the XML MIME types:

- `text/xml`
- `application/xml`
- Any mime types that ends with `+xml`

and if the file is less than or equal to the specified maximum size, then the crawler should retrieve the file and process it; otherwise it should ignore it and move on. For more details on XML media types, see RFC 3023 (http://www.ietf.org/rfc/rfc3023.txt). Your crawler should also not retrieve the file if it has not been modified since the last time it was crawled, but it should still process unchanged files (i.e., match them against XPaths and extract links from them) using the copy in its local database.

We have given you some "helper" classes in `edu.upenn.cis.cis455.crawler.crawler.utils` that might be useful to store information about URLs and robots.txt.

Certain web content, such as the papers in ACM's Digital Library, normally costs money to download but is free from Penn's campus network. If your crawler accidentally downloads a lot of this content, this will cause a lot of trouble. To prevent this, you **must** send the header `User-Agent: cis455crawler` in each request.

3.5. Test cases

You must develop at least two JUnit tests for storage system and two more for the crawler.

You should next **add a Route to enable retrieval of documents from the BDBstore**, using the following form:

```
localhost:45555/lookup?url=…
```

that takes a GET request with a URL as parameter `url`, and returns the stored document corresponding to that URL. Think of this as the equivalent of Google's cache. If the document was not crawled, your server should return a 404 error. We will use this interface for testing.

3.6. Milestone 1 Submission

Submit a zip file on Canvas as before.

4. Milestone 2: Streaming Crawler and Matching Engine

The next milestone will consist of an evaluator for *streams* of document content. You will extend your Milestone 1 project to run on a **stream processing engine** that enables multithreaded execution. You will also extend your application and storage system to enable users to register "subscription channels" with the system. Finally, you will build a stream processing component that checks documents against the various "channels" and outputs results per user.

4.1 Rework the Crawler as a "Spout," "Bolt," and Shared Modules in StormLite

Your Milestone 1 project had a simple execution model, in which you controlled the execution of the crawler and presumably did this in a crawler loop. Now we want to break in into a smaller unit of work that can be parallelized.

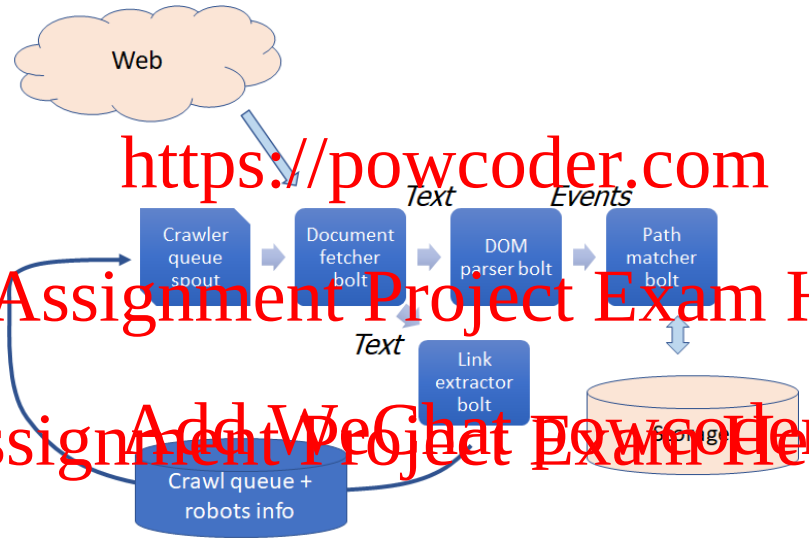
To do this, we'll be using a CIS 455/555-custom emulation of the Apache Storm stream engine, which we call **StormLite** (it should show up in your HW2 repo already). Please see the document on StormLite and see **TestWordCount** (in the test directory) as an example of a multithreaded stream job. Storm has **spouts** that produce data one result at a time, and **bolts** that process data one result at a time. As with our emulation of the Spark Framework, you should be able to use examples of Apache Storm code to understand how StormLite works.

You should **refactor your Milestone 1 crawl task** to run within StormLite, as follows. Note that StormLite supports multiple worker threads but you can control the number of "executors" (and start with 1).

1. You will maintain (or update) your frontier/crawl queue of URLs from Milestone 1. However, you want to place it in a "spout" (implement **IRichSpout**) so it sends URLs one at a time to the crawler via the **nextTuple** interface.
2. You will maintain your BerkeleyDB storage system from Milestone 1. This will also be a shared object, at least across some aspects of your Milestone 2 implementation. Again, you may want to use a "singleton factory" pattern.
3. The crawler should be placed in a **bolt** – its **execute** method gets called once for each URL from the crawler queue. The crawler should output documents one at a time to its output stream. See the **IRichBolt** interface and the example code.
4. Now, in our suggested (but not mandatory) architecture, there should be two "downstream" bolts that take documents. (It is perfectly possible to send an output stream to two destinations.)
5. Lower branch:
    a. One **bolt** should have an **execute** method that takes a document, writes it to the BerkeleyDB storage system, and outputs a stream of extracted URLs.
    b. Next, there should be a **bolt** that filters URLs (using appropriate techniques and data structures) and updates the shared frontier queue.
6. Upper branch:
    a. A second (in parallel) **bolt** should take a document and parse it using JSoup or another parser that takes into account element structure. It should send *streams* of

OccurrenceEvents. This bolt will traverse the entire DOM tree in the DOM, using a standard tree traversal. You will send an **OccurrenceEvent** each time you traverse to an element node from its parent (**ElementOpen**), each time you traverse to a text node (**Text**), or each time you traverse back up to an element node from its children (**ElementClose**).

b. Finally, there should be a **bolt** that checks for **matches to channels** by using the streams of events. When there is a match, it should update the BerkeleyDB store accordingly.



Based on the sample code in **test.upenn.cis.stormlite**, the StormLite document, and (in fact) the documentation on Storm available from Stack Overflow and the Web, you should be able to assemble a stream dataflow like the one illustrated above.

You can assume a single crawler queue bolt, but should look at how the **fieldGrouping**, **allGrouping**, and **shuffleGrouping** specifiers allow you to specify how data gets distributed when there are multiple "executors" such as multiple copies of the crawler, parser, etc.

4.2 Extended Routes-based Web Interface

For Milestone 2, you will also enhance the Web application to support the following functions for logged in users.

4.2.1. Channels

Now that you have users and HTML or XML, we want to "connect" users with "interesting" content. To do this, any logged in user will be able to create **channels**. A channel is a named pattern describing a class of documents. An example of a *channel* definition would be

```
sports : /rss/channel/title[contains(text(),"sports")]
```

and you can see an example of content that would match the channel at:
http://rss.nytimes.com/services/xml/rss/nyt/Sports.xml

Assume that channels and their names are global.

You should implement an interface to create a channel, as a GET call:

`localhost:45555/create/{name}?xpath={xpath-pattern}`

4.2.2. Updated Login Screen

As before, you should have a login/registration form at `localhost:45555/register`. Once a user is logged in, you should have a "home page" at `localhost:45555/`.

- List all channels available on the system, and for each
- Include a link to the documents matching each *channel*, which triggers the application at `localhost:45555/show?channel={name}`

Obviously, you will need to add some logic to the BerkeleyDB storage system to store user subscriptions and to store which documents correspond to a channel (see Section 3.2 for how this will be populated). How you implement most of the functionality of the Web interface is entirely up to you; we are just constraining the URL interfaces. To make things consistent across assignments, we are specifying how the channel must be displayed by the "show" request.

- For each channel, a `<div class="channelheader">` element around its header, with the string "`Channel name: `" followed by the name of the channel; and the string "`, created by: `" followed by the username of the user who created the channel.
- For each HTTP or XML (e.g., RSS) document that matched an XPath in the channel:
    - The string "`Crawled on: `" followed by a date in the same format as `2019-10-31T17:45:48`, i.e. `YYYY-MM-DDThh:mm:ss`, where the `T` is a separator between the day and the time.
    - The string "`Location: `" followed by the URL of the document.
    - A `<div class="document">` element with the contents of the document.
- If the channel does not exist, return a 404.

We expect this application to run on your application server from the HW1. If you did not complete the HW1, or for some other reason do not want to continue to use the application server that you wrote, you may continue to use Spark Java with no penalty.

4.3 Pattern Engine as a StormLite Bolt

You need to write a class `edu.upenn.cis455.xpathengine.XPathEngineImpl` that implements `edu.upenn.cis455.xpathengine.XPathEngine` (included with the code in Bitbucket), and evaluates a set of XPath expressions on the specified HTML or XML document. Both protocols can be handled similarly, except that HTML, unlike XML, is case insensitive. Once you have tested that individually, you will incorporate it into (call it from) a StormLite **bolt**. We will be focusing *only* on elements, sub-elements, and text nodes.

The implementation object (instance of `XPathEngineImpl`) should be created via the `XPathEngineFactory`. The `setXPaths` method gives the class a number of XPaths to evaluate. The `isValid(i)` method should return `false` if the *i*th XPath was invalid, and `true` otherwise. You should implement the **evaluateEvent()** method:

1. This takes an **OccurrenceEvent**, which will have a document ID and a "parse event."
2. Given a set of registered XPaths, if the document associated with the event has satisfied (at any point) the XPath, a bit corresponding to that XPath should be set.
3. Your XPathEngine will need to store state for each document, to monitor its progress. This state should be initialized when you first encounter a document with an **ElementOpen**; it should be updated each time you get an event, and it should remove the state once all elements in the document have been closed.

To make things simpler, we are supporting a very limited subset of XPath, as specified by the following grammar (modulo white space, which **your engine should ignore**):

XPath → *(/ step)+*

step → *nodename* | *nodename*`[`*test*`]`

test → `text() = "..."`

→ `contains(text(), "...")`

where `nodename` is a valid XML identifier, and "..." indicates a quoted string. This means that a query like `/db/record/name[text() = "Alice"]` is valid. Recall that if two separate bracketed conditions are imposed at the same step in a query, both must apply for a node to be in the answer set.

Below are some examples of valid XPaths that you need to support (not an exhaustive list):

```
/foo/bar/xyz
/xyz/abc[contains(text(),"someSubstring")]
/a/b/c[text()="theEntireText"]
/d/e/f/foo[text()="something"]/bar
/a/b/c[text() =   "whiteSpacesShouldNotMatter"]
```

You should be able to think about these kinds of XPaths as regular expressions over open and close events.

The stream of **OccurrenceEvents** will be coming from a separate parser bolt (in our standard architecture; you can diverge from this if you prefer). You will probably want to create a test bolt when developing the XPath engine. The easiest HTML/XML parser to use in Java is probably a DOM (Document Object Model) parser, e.g., the one from JSoup. Such a parser builds an in-memory data structure holding an entire HTML or XML document. From there, it is easy to walk the tree and output events. You can also look into SAX parsers.

Once your XPath engine works over individual documents, you'll want to write a StormLite **bolt** whose **execute()** method instantiates the XPath engine for a given input document (passed in as a tuple), looks up all of the **channels** defined in the BerkeleyDB database, and for each document that matches an XPath for a channel, records the document as a match to the channel. Subsequently, the Web application interface will be able to show the documents as matches.

4.4 Unit Tests

In addition, you must implement at least 5 *unit tests*, using the JUnit package (see the section on Testing below for helpful Web page references). JUnit provides automated test scaffolding for your code: you can set up a set of basic objects for all of the test cases, then have the test cases run one-by-one.

Your JUnit test suite should instantiate any necessary objects with some test data (e.g., parse a given HTML or XML document or build a DOM tree), then run a series of unit tests that validate your Web application and your XPath matcher. In total you must have at least 5 unit tests (perhaps each designed to exercise some particular functionality) and at least one must be for the Web application and one for the XPath evaluator.

4.5. Submitting Milestone 2

Your solution must meet the following requirements (please read carefully!):

1. You must implement the `edu.upenn.cis455.xpathengine.XPathEngine` interface.
2. Your XPath engine class must be must be created by the `XPathFactory` when the appropriate static method is called.
3. Your submission must contain a) the entire source code, as well as any supplementary files needed to build your solution, b) a working Maven `pom.xml` and/or c) a `README` file that contains 1) your full name and SEAS login name, 2) any extra credit claimed, 3) any special instructions for building or running.
4. Your code must contain a reasonable amount of useful documentation.

Reminder: All the code you submit (other than the dependencies on the JSoup/JTidy/TagSoup parser, the standard Java libraries, and any code we have provided) must have been written by you personally, and you may not collaborate with anyone else on this assignment. Copying code from the web is considered plagiarism.

5. Testing the Crawler

5.1. 'Sandbox'

We have implemented a small sandbox for you to test your code on. It runs on machines in Penn Engineering, so it will be fast to access, and it will not contain any links out of itself. The start URL of the sandbox is https://crawltest.cis.upenn.edu/. There should be adequate XML and HTML documents there to test your XPath matching.

5.2. JUnit

In order to encourage modularization and test driven development, you will be required to code test cases using the JUnit package (http://www.junit.org/) - a framework that allows you to write and run tests over your modules. A single test case consists of a class of methods, each of which (usually) tests one of your source classes and its methods. A test suite is a class that allows you to run all your test cases as a single program. You can get more information here: http://www.onjava.com/pub/a/onjava/2004/02/04/juie.html.

For Milestone 1, you must include 5 test cases and for Milestone 2, a test suite consisting of these 5 and at least 2 more for each of DOM Parser and Path Matcher (for a total of 5 new tests). If your test suite uses any files (e.g., test inputs), please put them into your project folder and use a relative path, so your tests will run correctly on the graders' machines.

6. Extra credit

There are several enhancements you can add to your assignment for extra credit. In all cases, if you implement an improved component, you do not need to implement the simpler version described above; however, your improved component must still pass our test suite for the basic version, and you will lose points if it does not. A safer approach is to implement the basic version first and then extend it later; if you choose to do this, and if you submit both versions, please document in your `README` file how to enable the extra functionality.

6.1 BYOF - Bring your own framework (+5%)

Rather than using Spark Framework, get your Homework 2 working on your Homework 1 Milestone 2 framework. From your Terminal, after 555-hw2 is pulled to your local repository, also run "git clone" (with your HW1 repo) to pull your **Homework 1 Milestone 2** code**.** Then go into the appropriate subdirectory for HW1 and run the following to put it into a local Maven repository:

```
mvn clean install
mvn deploy:deploy-file -Dfile=target/homework-1-1.0-SNAPSHOT.jar -
DpomFile=pom.xml -Durl=file:/vagrant/555-hw2/maven-repository/ -
DrepositoryId=maven-repository -DupdateReleaseInfo=true
```

Then modify your Homework 2 **pom.xml** to add:

```
<dependency>
      <groupId>edu.upenn.cis.cis455</groupId>
      <artifactId>homework-1</artifactId>
      <version>1.0-SNAPSHOT</version>
</dependency>
```

and of course, remove Spark Framework and change the imports from **spark.Spark.*** to the appropriate imports for your HW1.

6.2. Crawler web interface (+10%)

For 10% extra credit, provide a Web interface for the crawler. An admin user (not all users) should be able to start the crawler at a specific page, set crawler parameters, stop the crawler if it is running, and display statistics about the crawler's execution, such as

- the number of HTML pages scanned for links,
- the number of XML documents retrieved,
- the amount of data downloaded,
- the number of servers visited,
- the number of XML documents that match each channel, and
- the servers with the most XML documents that match one of the channels.

This will entail using some sort of communication between processes, potentially through the storage system or via an HTTP request.

6.3. User / channel subscriptions (+5%)

For 5% extra credit, allow users to choose which channels to subscribe to from a list of available channels. Add a list of the channels to which they are subscribed, in addition to all channels. Add a "subscription" interface at: `localhost:45555/subscribe?channel={name}`

Here, when a user logs in, you should only present content from the subscribed channels, as opposed to all channels.