CIS 455/555: Internet and Web Systems

Fall 2020

Team Project Specifications

Final code and **als week)**

1 Overview

For the term project, you will build a distributed Web indexer/crawler and analyze its performance. This will involve several components, each of which is loosely coupl

- Crawler
- Indexer/TF-IDF Retrieval Engine
- PageRank
- Search Engine
- Experimental

In addition, you may need to build a command-line in u to launch the components. More details on each component are provid latively open-ended and includes many possibilities for extra credit; the EC ite st suggestions, and you should feel free to provide other or additional features if you like. However, you are strongly encouraged to <u>get the basic functionality working first</u>!!

Suggested approach: Spend some time early coordinating with your group-mates and deciding which modules from your previous homework assignments are "best of breed." Designate one person to be responsible for each task.

Make sure adequate time is spent defining **interfaces between components** (in this case, appropriate interfaces might be REST-style messages analogous to HW3, Web service calls, and perhaps common index structures), and also plan to spend significant time integrating, **especially with EC2 and real data**. We strongly recommend that you do integration testing *regularly*, ideally every few days. Not only is this important to catch issues with compatibility, it is also helpful catch search result quality issues as well.

As in previous projects, please consider the use of automated tools for building your project (Maven) and for version control (Git, with a **private repository on BitBucket/GitHub/GitLab**).

Note that the report includes a non-trivial evaluation component.

You may want to make use of (1) Vagrant for debugging and development, (2) Bitbucket for sharing your work, (3) JUnit tests for validating that the code works (or remains working), (4) Amazon EC2 or Google Cloud to deploy and evaluate your system.

2 Project specifications

2.1 Crawler

The Web crawler should build upon your past homework assignments, and it should be able to parse typical HTML documents. It should check for and respect the restrictions in `robots.txt` and be well-behaved in terms of concurrently requ ... should be distributed, Mercator-style, across multi ... implementation from HW3 and your basic implem

Just as in HW2, the crawler should track visited pages and not index a page more than once. It also ***must*** identify itself as 'cis455crawler' in the User-Agent header!!

Recommendations: Try to start crawling early; your team will need at least a small corpus to test the other components! It is useful to keep much of the crawler's state (UR ... at you can interrupt a crawl when errors happen and continue it later on. Si ... o keep the crawled documents around in some sort of persistent storage so that y ... wl everything if something goes wrong. Please be careful not to DoS attack any

For best results, you sh ... me when it runs into spider traps or starts crawling lo ... 000,000 high-quality documents. One test is ... ify that your crawler gets to all of the regions that you expect in a timely fashion. Rem ... your team's indexing is, garbage in => garbage out.

Extra credit: Support crawling and indexing additional con ... ents, Word documents, etc. You can also provide a way to search for images, e.g., based on the anchor tags of links that point to them, the 'alt' text that is provided in the img tags, or words that may appear in their URLs. You may use Apache Tika or PDFBox as libraries for reading these file formats.

Extra credit: Extend your crawler to collect additional off-page features about the pages it crawls, e.g., the location of the server or the age/stability of the domain registration, and provide a way to use this information for ranking.

2.2 Indexer

The indexer should take words and other information from the crawler and create a **lexicon**, **inverted index**, and any other data structures that are necessary for answering queries. Your indexer should provide a way to return weighted answers that make use of TF/IDF, proximity, and any other ranking features that are appropriate.

It should use MapReduce or Storm/StormLite to generate the data structures from the output of the crawler, either using the **StormLite framework from HW3 or Apache Hadoop/Elastic MapReduce**. It should store the resulting index data persistently across multiple nodes using BerkeleyDB or a cloud storage system like S3, DynamoDB, or RDS so lookups are fast.

Recommendations: Have a look at the Google paper and consider adding some of the more advanced index features they have there, e.g., the distinction between normal, fancy, and anchor hits, or the support for phrase search. For the index computation itself (TF, IDF, hit lists, web link graph), we recommend that you write a few small MapReduce jobs. Since your index will be distributed across multiple nodes, you'll also need to provide some kind of interface for the scoring/ranking component to look up data in the index. One easy way to do this is to use REST-style messages, just like in HW3.

Extra credit: Include document and word metadata that might be useful in creating improved rankings (e.g., the context of the words -- show a small excerpt of the original document on your results page in which the hits are highlighted, e.g., in bold).

Extra credit: Provide locati e to infer that a page has a connection to a partic an often infer the location of the user that submits the q okup.

Extra credit: Fault tolerance. Your search engine should continue to work if some of the nodes crash or are terminated. This requires at least some degree of replication (so that parts of the index do not become unavailable when crashes occur) and a way to monitor the 'health' of the nodes in the index.

2.3 PageRank

Given information from crawling, you should perform **link analysis** using the PageRank algorithm, as discussed in class and i k as a MapReduce job.

Recommendations: G t first you'll need to address a number of ch ou have not yet crawled) or PageRank sinks, how to encode the data such that the out rve as input to the next, or how to test whether the PageRank values have co ty issues, it is a good idea to manually inspect some of the values you get, to see with your intuition of the pages' quality (e.g., Wikipedia vs. some little-know r team members have a sufficiently robust HW3 implementation, you may want to consider Elastic MapReduce.

Extra credit: Adapt your HW3 solution to support more efficient computation across multiple iterations. Since PageRank is an iterative algorithm, you should automatically run multiple instances of a job (otherwise you'll have to trigger each iteration manually in the web interface). However, you should also consider mechanisms for reusing results across iterations. Apache Spark (not Spark Java), discussed below, may serve as inspiration. There is also a research paper on “Haloop” that might give some ideas.

Extra credit: Use **Apache Spark**, a more modern “successor” to Hadoop MapReduce, as the basis of your PageRank implementation. While Spark implementations of PageRank exist, **you are expected to write your own implementation**, and you are expected to extend these with **support for dangling links, self-loops, etc**.

2.4 Search Engine and Web User Interface

This component is fairly self-explanatory, as the goal is to provide a search form and weighted results list. One aspect that will take some experimentation is determining how to combine the various factors (PageRank, TF/IDF, other word features) into a single ranking of the results. This user interface portion of this must make use of route handlers in Spark Framework or your HW1 server. You may leverage open-source client-side libraries such as Vue.js, Bootstrap, etc.

Recommendations: Ranking performance is crucial - the best scalable/secure/robust search engine design doesn't help if your search results are not useful! Therefore, you need to spend at least a few days on tuning your ranking function. This is challenging because the ranking component depends on all the other components, so you should insist that your teammates quickly provide simple but functional prototypes of their components (single-node crawler, basic indexer, basic PageRank, etc.), so that you can start testing.

It is also a good idea to have a 'debug mode' in which your engine shows additional information about how the results were ranked, and why (e.g., the raw PageRank scores, the TF and IDF values, and so on), so you can do example queries ("A ... te why the results you expected do not show up at o

Extra credit: Integrate sear ... Amazon or eBay, weather forecasts, business info from Yelp, results from Facebook, etc. Simple implementations could display these results in a sidebar or in a box above or below the other results; more advanced implementations could merge the external results with the search engine's own results (but please visibly indicate which is which!).

Extra credit: Implement a simple Google-style spell-check for ... simple edits to the word (e.g., adding, removing, transposing characters) and see if ... rd is "nearby."

Extra credit: Consider ... le example of this would be an autocomplete feature ... provide feedback about which entries are "goo

3 Experimental Analysis and Final Report

Building a Web system is clearly a very important and ch ... important is being able to convince others (your managers, instructors, peers) that you succeeded. We would like you to actually *evaluate* the performance of your methods, for instance relative to scalability.

For evaluation, you should log into multiple Amazon EC2 nodes and run the system.

One approach is to use the system with one, two, up to *n* nodes (where *n* is, say, 10 EC2 nodes), and compare overall performance, e.g., crawl throughput, time to run PageRank or to build the index, time to answer queries, etc. The result of such an experiment could be graphs that show some performance metric (e.g., crawl throughput) versus the number of nodes. For benchmarking query performance, you can write a simple query generating tool that submits many queries at once, and compare response time. What is the maximum number of concurrent requests you can reasonably handle, when varying the number of nodes? Can you separate out the overhead of the different components (including network traffic)? What is the bottleneck? Etc.

Your final report (a PDF document of six pages or less, due on the day of your demonstration) should include at least:

- Introduction: project goals, high-level approach, milestones, and division of labor

- Project architecture
- Implementation: non-trivial details
- Evaluation
- Conclusions

Note that the quality of the report *will* have substantial bearing on your grade: it is <u>not</u> simply something to be cobbled together at the last second!

4 Requirements

Your solution must meet the

1. Your **project plan** ... nvas. It must contain (1) the **first two sect** ... tion of the proposed project architecture, (2) some **rough milestones**, (3) your **private bitbucket repository** (for which you should grant read permissions to the email **upenncis455@gmail.com**), and (4) a list of team members with a **division of labor**. Create your project group on Canvas and submit the project plan there.

   a. To share the bitbucket repo, go to the repositor ... => “Repository settings” => “User and group access”. Then ad ... click “Add”.

   b. To create your project group, go to Canvas and click the “People” page => “Groups” tab. On this tab, you should be able to add yourself to a group. Note that some groups are 4-person and some are 5-person. **Add yourself to a group before submitting to ensure that everything is tracked correctly!**

2. Your **group check-in**, approximately 1.5-2 weeks after the project plan, will be with an assigned TA. Please message the TA directly to set up a 10 min slot for the check-in and **please do it early**. If you wait until the last day and the TA does not have time, that’s your fault not theirs. For this check-in, you should have made significant progress on every piece of your implementation including at least **50,000 web pages** crawled and a **usable webpage** that uses either mock data or (ideally) your integration testing data.

3. Your “code-complete” code submission, **due in the Bitbucket repository you list in your report**.

   The final version of the code submission, due with your demonstration, must be in the repository at that point. It must contain a) the entire source code for the project, as well as any supplementary files needed to build your solution, b) a README file, and c) the final report in point #5 below. The README file must contain 1) the full names and SEAS login names of all the project members, 2) a description of all features implemented, 3) any extra credit claimed, 4) a list of source files included, and 5) detailed instructions on how to install and run the project. The code must contain a reasonable amount of documentation. Please make this repository **private** and then share it with us.
4. Each team will need to **schedule a project demo** during the final exam period (or earlier if you prefer). All team m **e using your search engine on Amazon** **ust be able to access the search engine w**
5. The **final report**, du f six pages or less. It must include all the information from the project plan (possibly revised and/or with more details), plus a description of your implementation, your evaluation results, and your conclusions. **The final report should be submitted by one member of your team via Canvas.**

You may use the standard Java libraries, code from previous CI issions **by you or your team members**, and any code we provide or explicitly y code must be approved by us. Please don’t be tempted to reuse code from pa

To maintain fairness (a rces), we will create a Piazza post for approvi , you should post it there, along with a short desc e. We will then post a follow-up that indicate

5 Hints

Based on last year's project, many teams tend to divide up the work at the beginning and then meet again a few days before the deadline. This rarely works well; most people tend to under-estimate the amount of work that is needed to integrate the various components at the end. To avoid this, we recommend that you:

- Define clear interfaces at the beginning. Ideally, write a few unit tests, so that everyone under-stands what their component is supposed to do, and how it interacts with the other components.
- Do integration testing as early as possible. Ideally, everyone builds a very simple demo version of their component first, exchanges that code with the others, and then adds features one by one.
- Meet regularly and keep everyone posted on your progress.
- **Start early!!!**