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 coupled with the others:

- Crawler
- Indexer/TF-IDF Retrieval Engine
- PageRank
- Search Engine and User Interface
- Experimental Analysis and Final Report

In addition, you may need to build a command-line interface or an interactive menu to launch the components. More details on each component are provided below. The project is relatively open-ended and includes many possibilities for extra credit; the EC items we provide below are just suggestions, and you should feel free to provide other or additional features if you like. However, you are strongly encouraged to get the basic functionality working first!!

**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. We strongly recommend that you do integration testing *regularly*, ideally every few days. As in previous projects, please consider the use of automated tools for building your project (ant) and for version control (Git).

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

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

**Check/review sessions:** To ensure that each team is on track, each team must meet with one of the TAs during the week of April 4-8 and present a quick overview of their preliminary design and the intended division of labor. These sessions will not be graded; the goal is to get some initial feedback before the project plan is submitted. However, each TA can nominate one team for a bit of extra credit, based on the quality of the team's design.
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 requesting at most one document per hostname. Requests should be
distributed, Mercator-style, across multiple crawling peers, e.g., using your MapReduce framework from
HW3. (For instance, each node could locally collect the URLs it extracts from the crawled pages, and you
could change your master from HW3 to periodically run a MapReduce job that shuffles the URLs to the
node that is responsible for crawling them.) The crawler should track visited pages and not index a page
more than once. The crawler 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 with! It is useful to keep much of the crawler's state (URL queue, etc.) on disk, so that
you can interrupt a crawl when errors happen, and continue it later on. Similarly, it is a good idea to keep
the crawled documents around, so that you do not have to re-crawl everything if something goes wrong,
e.g., with your indexer. For best results, you should monitor your crawler carefully (to avoid wasting time
when it runs into spider traps or starts crawling low-quality pages) and aim for a corpus of at least
1,000,000 documents. Picking a set of good root URLs is also very important.

Extra credit: Add support for digests to detect when the same document has been retrieved more than
once, under different URLs. If so, the document should only be stored once – but two “hits” should be
returned.

Extra credit: Support crawling additional content types, e.g., PDF documents. 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.

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 to generate the data structures from the output of the crawler,
ideally using the framework from HW3 (but Hadoop/Elastic MapReduce are ok too), and it should store
the resulting index data persistently across multiple nodes using BerkeleyDB, 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 the /runmap etc. 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 location-specific results. In many cases you may be able to infer that a page has a connection to a particular location (e.g., based on addresses); similarly, you can often infer the location of the user that submits the query, e.g., via geolocation or by doing a reverse DNS lookup.

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 in the PageRank paper. You should implement PageRank as a MapReduce job.

Recommendations: Getting 'good' PageRank scores is harder than it may seem at first - you'll need to address a number of challenges, e.g., what to do with 'dangling links' (to pages you have not yet crawled) or PageRank sinks, how to encode the data such that the output of one iteration can serve as input to the next, or how to test whether the PageRank values have converged. To prevent quality issues, it is a good idea to manually inspect some of the values you get, to see how well they correspond with your intuition of the pages' quality (e.g., Wikipedia vs. some little-known web site). Since PageRank is an iterative algorithm, it may be useful to extend your HW3 solution to automatically run multiple instances of a job (otherwise you'll have to trigger each iteration manually in the web interface). If none of your team members have a sufficiently robust HW3 implementation, you may want to consider Elastic MapReduce.

Extra credit: SEO defenses. Look for signs that web sites are trying to boost their ranking, and find a way to adjust the PageRank values of these pages to account for this.

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.

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 ("Apple Computer", "Philadelphia subway", ...) and investigate why the results you expected do not show up at or near the top.

Extra credit: Integrate search results from web services, e.g., shopping results from Amazon or eBay, weather forecasts, business info from Yelp, 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 words with few hits, try simple edits to the word (e.g., adding, removing, transposing characters) and see if a much more popular word is "nearby."
Extra credit: Consider adding AJAX support to your search interface. A simple example of this would be an autocomplete feature; a more advanced example would be a way for users to provide feedback about which entries are “good” or “bad,” and use these to re-rank future results.

3 Experimental Analysis and Final Report

Building a Web system is clearly a very important and challenging task, but equally 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 May 6) 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 not simply something to be cobbled together at the last second!

4 Requirements

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

1. Your project plan, due on April 11, 2016, must be a PDF file of two pages or less. It must contain the first two sections of the final report, i.e., the introduction, a description of the project architecture, some rough milestones, and a division of labor. One team member must submit the project plan via the online submission system.
2. Your code submission, due on April 27, 2016, must contain a) the entire source code for the project, as well as any supplementary files needed to build your solution, and b) a README file. 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. The code must be submitted by one team member via the online submission system.
3. Each team will need to schedule a project demo during the finals period. A list of available time slots will be posted on Piazza. All team members must be physically present for the demo; absent team members will receive a zero.

4. The final report, due on May 6, 2016, must be a PDF file of 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. One project member must submit the report via the online submission system.

You may not use any third-party code other than the standard Java libraries, code from previous CIS455/555 homework submissions by you or your team members, and any code we provide or explicitly approve. To maintain fairness (and to ensure that all teams have access to the same resources), we will create a Piazza post for approving third-party tools; if you would like to use such a tool, you should post it there, along with a short description of what it does, and a URL to a relevant web page. We will then post a follow-up that indicates whether the tool may be used.

5 Hints
Based on my experience with 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 underestimate the amount of work that is needed to integrate the various components at the end. To avoid this, I recommend that you

- Define clear interfaces at the beginning. Ideally, write a few unit tests, so that everyone understands 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.
- Be sure to keep an eye on the services that are active on your AWS account! For instance, a few forgotten VMs that keep running for several days can quickly use up your AWS Educate credits. If you run out of credits, AWS will charge your credit card.
- **Start early!!!**