***Introduction***

We are implementing a distributed crawler and search engine based on the Mercator Crawler and the Brin and Page Google and PageRank papers. Our primary goals are, in roughly decreasing order of priority, to return reasonable and accurate results for a given search; to traverse a sufficient number of web pages to have comprehensive results for a variety of searches; to return results relatively quickly; and to have a user-friendly interface.

The primary person responsible for the crawler, indexer, PageRank, and search engine and web UI is, respectively, Josh Fried, Josh Kessler, Max Tromanhauser, and Ryan Smith.

Our initial approach to designing the crawler and search engine was to identify dependencies among the different components of the project and potential bottlenecks. Although some components may be run in parallel, others rely on the output of other components in the chain to begin. The crawler will produce documents and document IDs and store them in a database that will read by the indexer. The indexer will analyze the documents and store their associated information in a database that can be read by the PageRank process and the search engine. The PageRank process will read links stored by the indexer and calculate and store PageRank information. Finally, the search engine will use the results from each component to determine the suggestions to be presented.

We determined that a relatively small sample of documents would let us begin testing the indexer and PageRank, but it would be difficult to validate that we were weighting factors appropriately in our search engine until we had a corpus that was both adequately broad and deep. We also want to minimize the time spent transferring data, so we decided to use DynamoDB for its support of batch operations.

We are targeting a December 3rd “code complete” deadline, allowing us a week to integrate the components. We will also begin our final crawl upon reaching the first milestone, though earlier data will be used for testing other components and may be included in the final corpus. We aim to have refined our search weighting by December 9th.

***Crawler***

The crawler will use a highly multithreaded and highly distributed model. There will be a master daemon responsible for coordinating the various crawler nodes. Each crawler thread that runs will maintain a queue of domain names to cycle through, each of which will contain a set of sites to be crawled. Threads will be distributed across the machines and coordinated by the master daemon. Each domain will be mapped to a crawler based on its hash. The crawler will make use of a DynamoDB table to keep track of what sites have been crawled during which crawl session. Each crawl instance will have a thread that continuously records the state of the crawler to a disk so that it can be resumed on errors. Downloaded documents will be saved into a filesystem on an EBS volume attached to all crawler nodes and accessible by the indexer and pagerank pieces.

***Indexer***

The indexer will simultaneously read documents from DynamoDB and begin processing them. We are considering two approaches for the indexer. First, the crawler might notify the indexer when it has finished storing some number of documents in the database, and the indexer will then retrieve them. Alternatively, the indexer can be scheduled to check for updates independently at some regular interval. When the indexer reads documents, it will unbatch the documents and individually process them, using JSoup to extract: link metadata, possibly text-formatting data, such as whether some words are larger than others, and the non-HTML text of the document. It will then track word location (which can be used to enable phrase searching and showing short portions of the documents in our search results), after which MapReduce jobs will be run via Hadoop to create the inverted index.

***PageRank***

The PageRank process will read from the databases that the Indexer and Crawler are updating and issue MapReduce jobs to calculate each site’s pagerank. The MapReduce job will use parent document IDs as keys and the document IDs that correspond to outgoing links as values. In the interest of getting a prototype and data quickly, our initial plan is to have each MapReduce perform one iteration of PageRank rather than using “Hadoop” or other, smarter ways of doing multiple iterations. Depending on our results, we may look into using a less naive approach.

There are different options for handling dangling links, including combining all unexplored links into one document ID with links to all other pages, or simply ignoring their existence and removing their links from incident documents. PageRank sinks can be dealt with using a random jump probability that should keep sinks from ruining the system; initial research suggests one seventh is a good starting point. These solutions don’t seem particularly complex, and once we have results, we will be able to determine whether we need to adjust our strategy.

***Search Engine and User Interface***

The user interface will consist of a clean, simple search page similar to Google. It will be built using the basic servlet architecture used in our homework assignments. The servlets will provide search engine functionality that will determine which results should be returned. It will do this by evaluating the search query input (remove grammar, etc.); using data from the indexer in order to compile a list of URLs and calculate tf-idf scores; fetching those URLs’ PageRank scores; and ordering the results based on some weighting the tf-idf scores and PageRank. It will prioritize relevant, fast results. If time allows, we aim to include:

1. A caching layer between the search engine servlets and the indexer and PageRank components. This would be targeted at the most frequently searched words and the most frequently returned URLs. A possible opensource tool that might be used is Memcached.
2. Phrase search, using term location information from the indexer.
3. The User Interface may use opensource libraries to enable pagination of results, which will improve the user experience and limit load times by only loading the data needed.
4. The search form will use an outside API for predictive search term completion. This will benefit the project by reducing the number of grammatical errors and superficial variations of similar search queries that may cause inefficiency in caching.
5. A threshold will be set for limited results. If a search term receives very few results, the search engine will attempt to determine a more productive search query by conducting very limited NLP. For example, “Best food Philedelfia” may become “Best food Philadelphia”.
6. User interface will support a page preview. This will most likely be completed by opening a simple iframe to a given URL via Ajax calls on mouse hover.
7. A debug mode will be built in so that interested users (as well as the team during development stage) can see how results were ranked.