Stock News Indexer

Table of Contents

1. Introduction
2. Motivation
3. Architecture
4. Issues
5. Areas for Improvement
6. Conclusion

Introduction

The purpose of this project was to build a ranked, stock news indexer and management console in an effort to test my understanding of perl, learn basic socket programming and the HTTP protocol, and experiment with input manipulation on the web.

Motivation

The choice to focus on indexing stock market news was based on a long, personal interest in the stock market. Knowing that the market is not a zero-sum game, I've contemplated several times building a stock analysis tool for assisting with trading decisions. Many programmers and traders have had similar ambitions and there currently exist many rich analysis tools that far surpass anything I may ever build. Nevertheless, I chose this opportunity to build a stock news indexer as a building block for a hypothetical trading tool I may or may not build in the future.

Architecture

The project consists of four components: a datastore, indexer, worker, and web interface.

Datastore

The datastore I chose for this project was Redis, a highly-performant key/value store that supports lists, sets, hashes, sorted sets, and strings. I was introduced to Redis superficially in the Cloud Database course and knew the interface to be short, clean and easy to understand. I chose this datastore for several reasons, the first being that this seemed like a great opportunity to play with it. But the main reason was I felt the tool mapped well to the data I needed to store. The data with which I'd be mainly working was just a set of urls keyed by a stock symbol, e.g. "MSFT" => [url1, url2, url3]. A relational database seemed heavy-handed but I also needed more than I thought DBM could provide. I also wanted something I could interface with using the web interface I was planning on building.

Indexer

The indexer is a simple, robots.txt-obeying, forking, recursive indexer. The indexer takes as an argument a stock symbol and for each source in the list of sources, forks a child and begins spidering each source. If the symbol is found on the page, the indexer stores the url for the page and logs some statistics. The recusion depth for testing and demo was set to 2 levels, but nothing prevents it from being set higher.

I played around with several ways to make the indexer more performant, including threads, async operations, and eventually settled on forking because it was simple to implement and performed the best.

The main spider function works as follows. Given a url and stock symbol, the indexer initially does some very basic error checking - return if the url is nil, an empty string, matches on "doubleclick", etc. Doing so prevents us from indexing content served via ad networks, empty links, and the like. Next we make our request, checking Redis for a cached version of the robots.txt content for the url host along the way. If it exists, we continue. If not, we fetch it and cache it for use later. If the request response is a 200, we strip the page of script and style tags, then search the page for the supplied ticker argument. If we get a match, we log a "hit", else we log a "miss". Next we collect all links on the page and call the same function again for each link. Using the just-mentioned basic statistics, the indexer will actually dynamically build a set a sources when the number of "hits" becomes greater than some user defined threshold.

Doing so allows the indexer to discover new sources to be used the next time it begins indexing.

Worker

So we can index more than one stock symbol at a time, I created a 'work' queue in Redis to be polled by a worker I wrote in Perl. The worker loops indefintely, checking the queue in Redis every two seconds. Jobs are pushed onto the queue either directly via redis-cli or via the web interface. If a job is found, the worker "shells out" to the indexer to process the job. I chose to wrap this in an Async block so that we return immediately, allowing us to fetch more work from the queue and index multiple tickers are the same time.

Web Interface

A web interface was written to explore several concepts touched on briefly during class including dynamic HTML generation and submitting content to a server via web forms. Doing so allowed me to become more familiar with those tools but also build an administrative interface for managing the indexer.

The web interface was created using Sinatra, a minimal Ruby web framework, and Twitter Bootstrap, an HTML, CSS, and Javascript framework that supplies some easy-to-use and consistent styling.

The GUI consists of four main pages. The home page contains some basic metrics about how many articles have been found at each source and a ranked list of stock symbols with the most indexed pages. The Tickers page contains a list of stock tickers and article counts. It also contains a bevy of controls for adding a new symbol to be indexed, reindexing individual tickers, or reindexing all symbols. Each "detail" view for a ticker contains a list of indexed pages. The sources page allows you to add new sources manually. Finally, the Queue page allows you to inspect jobs in the queue to be processed. This view is largely unused as the worker digests jobs very quickly, but it does serve as a sanity check to confirm whether jobs are actually being popped off the queue.

Web forms were written without javascript using vanilla HTML. Server actions were defined using the Sinatra DSL in app.rb

Issues

Like all hastily built software, this project contains several notable bugs and features that leave one wanting.

The most glaring bug is the simplicity with which the indexer searches for a symbol on a page. The indexer checks for a match against the entire source of the page, but only in a very rudimenary fashion - `if($body =~ /\W($ticker)\W/ig)`. This searches the entire HTML source and matches on the symbol surrounded by any non-word character, which tends to work pretty well. The downside to this strategy is that it matches on inline javascript variables and functions like " FB.init".

Another issue I noticed was when searching for the stock ticker "FB" (Facebook). I do discard script and style tags before matching but I noticed many Facebook-centric HTML id's and classes are prefixed with the string "fb", meaning we sometimes get false-positives. Noticing this I toyed with the idea of selectively searching content, rather than attempting to filter content. A selective search strategy would select only p, h1, h2, and span tags, for instance and match against those.

Areas for Improvement

The largest area for improvement would be in what types of content from the response body are searched. As just mentioned above, with a bit more time, effort, and/or looser restrictions on modules, I would have been able to match against just textual content - p tags, title tags, h1/h2/h3 tags, etc. - either by filtering out content or only searcing certain tags. This would have greatly improved accuracy of the indexer.

The next area I would have loved to improve was the dynamic source list. Currently the indexer logs hits and misses and addes "hot" hosts to the list of sources. However I never built into the indexer an expiration mechanism for sources that took advantage of the misses. Hits and misses also aren't reset at any point in the operation of the indexer except manually. It would have been nice to display a graph for each host that plotted hits vs. misses.

Another way to improve the indexer would be to fully index each page. Most indexers do more than match on a single search term, and when they do match, they usually compute or build a list of stemmed words contained on the page,

excluding a myriad of stop-words like "is", "the", and "a". Doing some might allow us to do some background processing to cleanup the search results or even auto-summarize the content.

One nice-to-have feature is a page blacklister. While outside the scope of this project, it would have been neat if the indexer would have recognized familiar content and blacklisted either blocks of content or entire pages. For instance, a "Top 10 Most Volatile" widget may appear in the top right corner of every page. If the ticker for which you're indexing is in that list, you may end up storing every single page of a host.

And finally, pages need TTL's associated with them. Pages are taken down, updated, or replaced with new pages.

Conclusion

I encountered very few road blocks during this project. I genuinely enjoyed building this indexer and believe I accomplished my goals of learning basic socket programming, building a system for indexing stock news, and building on my knowldge of web forms. I identified a number of improvements that could be made to the indexer, most of which are relatively easy to implement. Development of the web console gave me some great insight into exactly what content the indexer was indexing. In the future however I may focus more on measurement and analytics, as the indexer would have truly shined had I been able to rank hosts as valuable or not valuable sources for stock news.