Code-related text search engine

Report

Yaveyn Anna
Simiyutin Boris

1 DESCRIPTION

Our project is a developer-oriented search engine. The main goal is to help people with technical problems, in a way similar to StackOverflow. Here is the link to project's repository.

2 ARCHITECTURE AND DESIGN

We have selected Kotlin as our main programming language. Although, we keep our system modular and use other languages when it is convenient. For instance document processing and invertid index building is implemented in Python.

For parallel crawling we use Akka actor system. Also we use Apache Spark framework for faster index building.

2.1 Data acquisition

2.1.1 General info. Data acquisition in our system is performed by crawler, which collects data from the Internet, starting from popular programmer-oriented sites, such as www.linux.org, stackoverflow.com, cppreference.com and docs.oracle.com. We use single configuration for every seed.

Our program meets politeness requirements for a web-crawler. We use open source RobotsTxt library for parsing robots.txt files and follow request rate conventions by 20 sec timeouts.

During development of crawler we faced performance issues, which were solved by making it multithreaded. We also had some troubles with intense memory usage. We fixed them by setting maximum total number of processed pages by each worker, which limited maximum size of it's page queue.

2.1.2 Concurrency. We decided to perform data acquisition in parallel and we chose an actor-based concurrency model to do so. The main advantage of using actors is that you can write highly concurrent, distributed and fault-tolerant code without thinking about synchronization between different threads. One of the most popular toolkit for actors in JVM is Akka - a cross-platform open-source framework written in Scala. We use it because it is a well-known and powerful tool with comprehensive documentation and large community.

The crawler consists of one manager actor and ten worker actors. Worker actors perform the actual crawling and the manager actor handles the communication between them. Each worker is responsible for it's own range of host names and holds it's own queue of Urls. When worker comes across a host name which it is not responsible for worker sends request to manager and manager redirects the Url to corresponding worker. The manager is also responsible for assigning workers for new host names.

2.1.3 Summary. The resulting program runs pretty fast and uses network channel rather efficiently. At first stage we crawled 103000 documents. However, we have some problems with rate limits at multi-domain resources e.g. stackexchange.org which we are going to fix on the next stage of the development.

2.2 Data storage

2.2.1 General info. This part of project is written in Python as there is a wide variety of python modules for text processing. We use Apache Spark as a framework for distributed indexing because it has convenient Python API.

2.2.2 Preprocessing. At first, we parse html code of pages and extract headers and body text for weighted ranking. Then, we remove punctuation, tokenise sentences, filter words from English dictionary using *enchant* module, remove stop words and perform stemming using *nltk* module.

2.2.3 Indexing. For each word and each file we compute the number of occurrences of this word in each header and in the body of a web page. For faster computation we use map-reduce technique which is implemented with the help of Apache Spark framework. As a result we have an inverted index which takes into account the fact that words from different parts of HTML document should be counted separately.

During the parsing of the crawled documents it turned out that some of the file names in our dataset is unacceptable for Apache Spark due to occurrence of some special symbols. Because of this problem we were forced to rename our documents but we kept links to initial web pages embedded into these documents so we do not lose the information about theirs origin. Also we created a table which maps new document names to corresponding URL's.

Index is stored as JSONed python dictionary. We expect full index file to be less than 600 MB so it's OK to store full index file in memory. Search will take constant time due to dictionary implementation.

2.2.4 Summary. We ran our indexer on subset of crawled data consisted of 5k pages. It took 8 minutes to process them and index file size is 30 MB. Extrapolating it, we suppose that it will take 3 hours and no more than 600 MB for building and storing index for full 100K dataset.