# Web Information Extraction and Retrieval
# Programming Assignment 1:
# Crawler implementation

Marko Prelevikj
63130345
mp2638@student.uni-lj.si

Gojko Hajduković
63180431
gh8590@student.uni-lj.si

Stefan Ivanišević
63170405
si0539@student.uni-lj.si

April 2019

## 1 Introduction

## 2 Implementation specifics

The implementation of the web crawler is done in Scala, which is a functional programming language, offering a lot of syntactical sugars which make the development process easier. We chose it in order to improve our programming skills in Scala, and also, learn a lot about its capabilities for concurrent programming.

### 2.1 Dependencies

To make the development process as easy as possible, we are using a number of dependencies. In this report, we are listing the once which are most significant, whilst the list of entire dependencies is available in the `build.sbt` file.

- `akka-actor` - providing us with the support of multi-threading through the concept of actor systems [1].
- `slick` - a functional relational mapping library to easily store data into the database [2].
- `crawler-commons` - a library containing common utilities for crawlers [3].
- `htmlunit` - headless browser which renders the html content of a provided URL [4].
- `bigqueue` - a multithread-safe persistent queue for keeping the frontier [5].
- `JSoup` - a library for HTML document parsing [6].

## 2.2 Database modifications

In order to make the implementation more insightful, we expanded the initial database with additional columns as follows:

### 2.2.1 Table `page`

We introduced the fields:

- `hash` - SHA256 hash of the entire HTML content of the page, used for duplicate detection.
- `load_time` - time needed to load the page
- `stored_time` - when the page was added in the queue

### 2.2.2 Table `page_type`

We introduced the following values:

- `INVALID` - in case there has occurred an unknown error while loading the page
- `DISALLOWED` - if the page is not allowed by the `robots.txt` file

### 2.2.3 Table `page_data`

We introduced the following column:

- `filename` - canonical url of the stored file

### 2.2.4 Storing process

We altered the storing process as well. We omit a page entry of the type *BINARY* which should be a reference to the *image* or *page_data* table. Instead, we are linking the resources directly to the pages where they occurred. For example, if there had been an image linked to a page with `id = 1`, we enter an image entry with a reference to the page with `id = 1`.

# 3 Crawler implementation

The development process of the crawler was done in multiple iterations. In the first iteration, we developed all the required utilities to build the crawler upon. These utilities include: URL-canonicalization, `SiteMap` parsing, `robots.txt` parsing, database service to store the obtained data. Furthermore, we developed the core concepts, and the basic pipeline of how the crawler should interact with the frontier and the database. Finally, we created workers which are going to perform the crawling.

There are two versions of the crawler workers. First, we tried a naive implementation in 3.2 with jittered time between the requests to the same domain, which didn't turn out well. Next, we developed a more advanced version of the worker in 3.3, which implements a distributed breadth-first approach.

### 3.1 Core concepts

The following steps describe the process of fetching and storing the data retrieved from a given URL. Whenever we refer to processing of a page, we are referring to the following steps:

1. frontier dequeueing - get the next page from the frontier
2. `robots.txt` check - check whether the page is allowed in the `robots.txt`, skip it if not. If the `robots.txt` is missing - allow it by default
3. page rendering - get the page content and HTTP status code using `HtmlUnit`
4. data extraction - extract all the detected links pages and binary data in the page using `JSoup`
5. data deduplication - detect the entries which already exist, and link them accordingly.
6. data storage - The duplicate detection is performed on a database level: if the URL exists - it is a duplicate, if it does not, it checks whether its hash code already exists, if it does not it is finally written into the database.
7. frontier enqueueing - All the non-duplicate links, images, and binary data is enqueued in the frontier to be processed when they come in line.
8. delay - after all the processing has been performed, the worker waits for at least `5s` until the next page is processed, depending on the presence of `robots.txt`.
9. repeat the process for the rest of the entries

### 3.2 Version 1 - Jittered Delay

Our naive approach is consisted of having multiple workers, which are reading from the frontier and processing the upcoming page. Each of the workers are spawned with a $5s$ jittered delay, where the jittering is in the range from 2 to 20 seconds.

We introduced the jitter in order to reduce the probability of having the crawler processing a page from the same domain. Unfortunately, since the crawler is working in a breadth-first manner, it is highly unlikely to have two sequential pages from two different domains.

Since we are limited to a single host machine, which has a single IP address, it would mean that we need to synchronise all threads and have a semaphore which will determine the delay between the requests.

The low level of concurrency would effectively mean that the multi-threading is not as effective as it could be if the concurrency was independent among the threads. Which lead us to the better approach, described in the section below. We illustrated our approach is illustrated in Figure 1.

### 3.3 Version 2 - Distributed Breadth-First

To take full advantage of the concurrency we decided to distribute the frontier among the pool of workers. The distribution is done such that each worker is in charge of a single domain - a *Domain Worker*, with the assumption that each of the domains has its own server. We illustrate this approach in Figure 2.

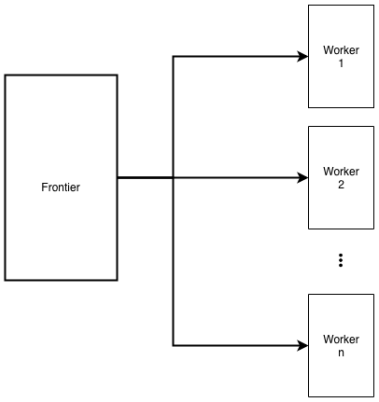


Figure 1: v.1.0 of the worker with a simultaneous access to the global frontier.

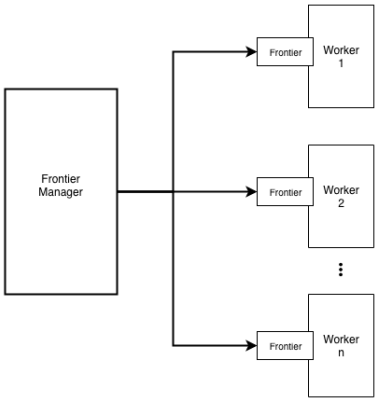


Figure 2: v2.0 of the worker with a *Frontier Manager* which delegates the workload.

To achieve the distribution, we introduced a special kind of worker which we call a *Frontier Manager*. Its purpose is to orchestrate the *domain workers* and route the workload to each of the workers.

By orchestration, we mean that the *Frontier Manager* has the permission to spawn new workers, if there are more domains available than workers currently. Another very important task it has is to pick a new domain for the worker which has finished processing a given domain. This is a crucial task, as it keeps all of the resources at full capacity and therefore it can maximize the number of pages it processes in a given time period.

Additionally, the *Frontier Manager* reroutes all the obtained links to the correct worker's frontier to be processed. If the given domain is currently inactive, then it keeps the pages in memory until it is chosen to be processed, when forwards them to the worker, and it resets the local state.

The *Domain Worker* is checking its status every $15s$ and if the local frontier is empty, it sends a signal to the *Frontier Manager* to get a new domain to process next. The delayed time between subsequent requests to the targetted server is not wasted by the worker, as it uses that time to enqueue all the incoming links from the *Frontier Manager*, and to check its own status (this is implemented as a message to the standard output), and to act accordingly.

# 4 Data Analysis

## 4.1 Experiment 1 - Provided seed all data

## 4.2 Experiment 2 - Extended seed all data

1

| | Original seed | Extended seed |
|---|---|---|
| running time [min] | 618 | 686 |
| avg wait time [min] | 111.057 | 119.629 |
| avg load time [ms] | 1351.347 | 1759.769 |
| sites | 288 | 327 |
| pages | 38268 | 58060 |
| duplicates | 971 | 1766 |
| links | 1585648 | 2569510 |
| images | 94368 | 187924 |
| avg images per page | 6.3567 | 8.520 |
| duplicate images | 64638 | 155277 |
| data | 28721 | 44761 |
| avg data per page | 4.9853 | 6.170 |
| duplicate data | 11688 | 12593 |
| doc | 32 | 226 |
| docx | 2 | 132 |
| pdf | 230 | 941 |

Table 1: General statistics

# 5 Conclusion

# References


[1] Lightbend, *Akka Documentation: Actor Systems*. [Online]. Available: https://doc.akka.io/docs/akka/current/general/actor-systems.html

[2] ——, *Slick*. [Online]. Available: http://slick.lightbend.com/doc/3.3.0/

[3] Crawler-Commons, *Crawler-Commons documentation*. [Online]. Available: http://crawler-commons.github.io/crawler-commons/1.0/

[4] HtmlUnit, *HtmlUnit documentation*. [Online]. Available: http://htmlunit.sourceforge.net/

[5] bulldog2011, *BigQueue documentation*. [Online]. Available: https://github.com/bulldog2011/bigqueue

[6] JSoup, *JSoup documentation*. [Online]. Available: https://jsoup.org/

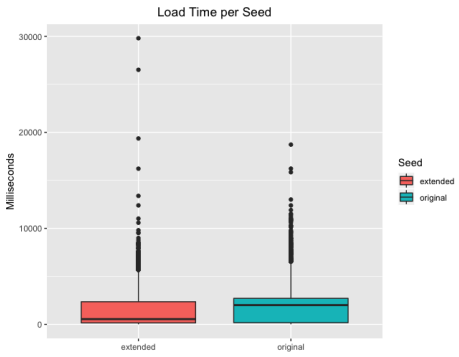


Figure 3: Load time.

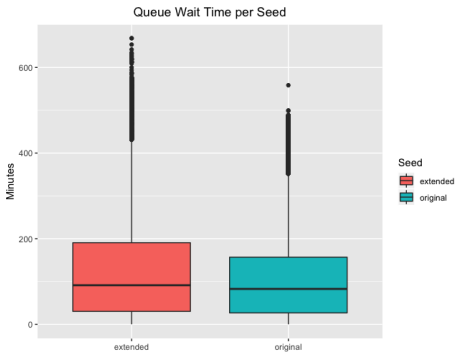


Figure 4: Queue wait time.