ACHE Crawler Documentation

Release 0.13.0

New York University

Jan 07, 2021

Contents:

1 Installation **3**
1.1 Using Docker 3
1.2 Build from source with Gradle 4
1.3 Download with Conda 4

2 Running a Focused Crawl **5**

3 Running a In-Depth Website Crawl **7**

4 Running a In-Depth Website Crawl with Cookies **9**

5 Crawling Dark Web Sites on the TOR network **11**

6 Target Page Classifiers **15**
6.1 Configuring Page Classifiers 15
6.2 Testing Page Classifiers 18

7 Crawling Strategies **19**
7.1 Scope 19
7.2 Hard-focus vs. Soft-focus 19
7.3 Link Classifiers 20
7.4 Online Learning 20
7.5 Backlink/Bipartite Crawling 21

8 Data Formats **23**
8.1 FILESYSTEM_* 23
8.2 FILES 24
8.3 WARC 24
8.4 ELASTICSEARCH 24
8.5 KAFKA 26

9 Link Filters **27**
9.1 Configuring using YAML 27
9.2 Configuring using .txt files 28

10 Web Server & REST API **29**
10.1 Security 29
10.2 Server Mode 29

10.3 API Endpoints . . . 30

11 HTTP Fetchers **35**
11.1 Switching the HTTP fetcher implementation . . . 35
11.2 Setting up a proxy . . . 35
11.3 Setting up a TOR proxy . . . 36

12 SeedFinder Tool **37**

13 Frequently Asked Questions **39**
13.1 What is inside the output directory? . . . 39
13.2 When will the crawler stop? . . . 39
13.3 How to limit the number of visited pages? . . . 39
13.4 What format is used to store crawled data? . . . 40
13.5 How can I save irrelevant pages? . . . 40
13.6 Does ACHE crawl webpages in languages other than English? . . . 40
13.7 Is there any limit on number of crawled webpages per website? . . . 40
13.8 Why am I getting a *SSL Handshake Exception* for some sites? . . . 40
13.9 Why am I getting a *SSL Protocol Exception* for some sites? . . . 40
13.10 Where to report bugs? . . . 40

14 Links **41**

HTTP Routing Table **43**

ACHE is a focused web crawler. It collects web pages that satisfy some specific criteria, e.g., pages that belong to a given domain or that contain a user-specified pattern. ACHE differs from generic crawlers in sense that it uses *page classifiers* to distinguish between relevant and irrelevant pages in a given domain. A page classifier can be defined as a simple regular expression (e.g., that matches every page that contains a specific word) or a machine-learning-based classification model. ACHE also automatically learns how to prioritize links in order to efficiently locate relevant content while avoiding the retrieval of irrelevant pages. While ACHE was originally designed to perform focused crawls, it also supports other crawling tasks, including crawling all pages in a given web site and crawling Dark Web sites (using the TOR protocol).

ACHE supports many features, such as:

- Regular crawling of a fixed list of web sites
- Discovery and crawling of new relevant web sites through automatic link prioritization
- Configuration of different types of pages classifiers (machine-learning, regex, etc.)
- Continuous re-crawling of sitemaps to discover new pages
- Indexing of crawled pages using Elasticsearch
- Web interface for searching crawled pages in real-time
- REST API and web-based user interface for crawler monitoring
- Crawling of hidden services using TOR proxies

CHAPTER 1

Installation

You can either build ACHE from the source code, download the executable binary using Conda, or use Docker to build an image and run ACHE in a container.

1.1 Using Docker

Prerequisite: You will need to install a recent version of Docker. See https://docs.docker.com/engine/installation/ for details on how to install Docker for your platform.

To use ACHE with Docker, you can 1) use a pre-built image or 2) build the image yourself as follows:

1. Using the pre-build docker image

We publish pre-built docker images on Docker Hub for each released version. You can run the latest image using:

```
docker run -p 8080:8080 vidanyu/ache:latest
```

Docker will automatically download the image from DockerHub and run it.

2. Build the image on your machine

Alternatively, you can build the image yourself and run it:

```
git clone https://github.com/ViDA-NYU/ache.git
cd ache
docker build -t ache .
```

where `ache` is the name of the image being built.

Running the crawler using Docker

The Dockerfile used to build the image exposes two data volumes so that you can mount a directory with your configuration files (at `/config`) and preserve the data stored by the crawler (at `/data`) after the container stops. In order to run ACHE using docker, you will need a command like:

```
docker run -v $PWD:/config -v $PWD/data:/data -p 8080:8080 vidanyu/ache startCrawl -c␣
↪/config/ -s /config/seeds.txt -o /data/
```

where `$PWD` is the path where your config file `ache.yml` and the `seeds.txt` are located and `$PWD/data` is the path where the crawled data will be stored. In this command `vidanyu/ache` refers to the pre-built image on DockerHub. If you built the image yourself, you should use the same name that you used to build the image.

1.2 Build from source with Gradle

Prerequisite: You will need to install recent version of Java (JDK 8 or latest) and Git.

To build ACHE from source, you can run the following commands in your terminal:

```
git clone https://github.com/ViDA-NYU/ache.git
cd ache
./gradlew installDist
```

which will generate an installation package under `ache/build/install/`. You can then make `ache` command available in the terminal by adding ACHE binaries to the `PATH` environment variable:

```
export ACHE_HOME="{path-to-cloned-ache-repository}/ache/build/install/ache"
export PATH="$ACHE_HOME/bin:$PATH"
```

This configuration will not persist after system restarts. To make it persistent, you will need configure the system to reload these settings automatically. Every operating system is configured in a different way. Following, is an example of how to install ACHE at `/opt` for Linux (tested only in **Ubuntu 16.04**):

```
sudo mv ache/build/install/ache /opt/
echo 'export ACHE_HOME="/opt/ache"' | sudo tee -a /etc/profile.d/ache.sh
echo 'export PATH="$ACHE_HOME/bin:$PATH"' | sudo tee -a /etc/profile.d/ache.sh
source /etc/profile.d/ache.sh
```

After this, the command `ache` will be available on the terminal, so you can simply run the crawler with the appropriate parameters.

1.3 Download with Conda

Prerequisite: You need to have Conda package manager installed in your system.

If you use the Conda package manager, you can install `ache` from Anaconda Cloud by running:

```
conda install -c vida-nyu ache
```

Warning: Only released tagged versions are published to Anaconda Cloud, so the version available through Conda may not be up-to-date. If you want to try the most recent version, please clone the repository and build from source or use the Docker version.

CHAPTER 2

Running a Focused Crawl

A focused (or topical) crawler crawls the Web in search of pages that belong to a given topic (or domain). To run a focused crawl using ACHE, you need to provide:

- Samples of relevant and irrelevant pages: ACHE analyzes these pages, and learns a classification model that is able to determine the relevance of the pages retrieved during the crawl.
- A list of seed URLs, which will be the starting point for the crawl. The seeds are just HTTP or HTTPS links of pages that are relevant to the topic – they can be the same as relevant pages supplied during model generation. ACHE will start crawling the Web from these links.
- A config file that provides settings/options that are used during the crawl.

In what follows, we provide details on how to run a focused crawl using ACHE.

1. **Build a Classification Model** Ideally, there should be as many relevant pages as irrelevant ones for training a good classification model. You should try to get as close as possible to a ratio of irrelevant to relevant pages of 1:1. If it's hard to collect relevant pages, a ratio of 5:1 would suffice. Another good practice is to collect samples of relevant and irrelevant pages for every web site. Collecting only relevant pages from a web site may lead to a model that classifies *any* page from that web site as relevant, which may not be appropriate.

   In order to collect the relevant and irrelevant pages, you can use a web-based system like the Domain Discovery Tool (DDT). DDT provides tools that streamline the process of gathering the training pages and building a model for ACHE. Alternatively, you can build the training set manually and build the model using ACHE's command line tool `ache buildModel`, as we describe below.

   Attention: You do NOT need to follow these instructions if you're using the Domain Discovery Tool (DDT). From DDT, you can create and save the model in an appropriate folder. Then, you can provide this folder as an input to ACHE.

   As you collect the training data, make sure the HTML files are organized in a directory with relevant pages in a folder named `positive` and the irrelevant ones in a folder named `negative`). See the directory config/sample_training_data in ACHE's repository for an example.

Optionaly, you can provide a file containing *stopwords* to the model-builder as well. Stopwords are words which contain little or no information about the context of the page. Some examples include: “the”, “at”, and “a”. A sample file is included in the repository as well at config/sample_config/stoplist.txt. If you don’t provide a stopwords file, a default list of common english stopwords bundled into ACHE will be used.

Finally, type in the terminal:

```
ache buildModel -c <Path of Stoplist> -t <Path of training data folder> -o
↪<Path to save model at>
```

This command should take a while to execute, depending on how many training samples you collected. In the end, it will print accuracy statistics about the derived model.

2. **Find Seeds** ACHE needs a list of URLs that will serve as starting points for the crawl. You can use as seeds the list of relevant pages generated in the previous step. A sample seed file can be found at config/sample.seeds. You can use the *SeedFinder* to help with this step as well. The SeedFinder uses commercial search engines to automatically discover a large set of seed URLs.

3. **Create Configuration File** To run ACHE, a configuration file is needed that contains the settings and rules the crawler will adhere to. A config file sample is provided in config/config_focused_crawl/ache.yml containing the following important parameters:

   - `link_storage.max_pages_per_domain` - The maximum number of pages to download per domain. This is useful to deal with crawler traps and redirects.
   - `link_storage.link_strategy.use_scope` - Whether the crawler should crawl the websites provided as seeds only. This needs to be set to `false` in a focused crawl.
   - `link_storage.online_learning.enabled` - Whether to enable relearning of the link classifier while the crawler is running.
   - `link_storage.online_learning.type` - Type of online learning to use. More details *here*.
   - `link_storage.scheduler.host_min_access_interval` - Time delay between sending subsequent requests to the same host. Should not be too small to avoid overloading web servers.

4. **Run ACHE** Finally, when you have created the *model*, the config file *ache.yml*, and the *seeds file*, you can run ACHE in the terminal:

```
ache startCrawl -c <Path of Config Folder> -o <Path of Output Folder> -s
↪<Path of Seeds File> -m <Path of Model Folder>
```

ACHE will continue crawling until interrupted using `CTRL+C`.

For large crawls, you should consider using `nohup` for running the process in background:

```
nohup ache startCrawl -c <Path of Config Folder> -o <Path of Output Folder> -
↪s <Path of Seeds File> -m <Path of Model Folder> > crawler-log.txt &
```

If you are running ACHE from **Docker**, you should use the following command:

```
docker run -v $CONFIG:/config -v $DATA:/data -p 8080:8080 vidanyu/ache:latest␣
↪startCrawl -c /config/ -s /config/seeds.txt -o /data/ -m /config/model/
```

where `$DATA` is the directory where the crawled data will be stored, and `$CONFIG` is the directory where the `ache.yml`, the `seeds.txt` file, and a directory named `model` (which contains the page classification model) are located.

CHAPTER 3

Running a In-Depth Website Crawl

When you already know sites you want to crawl, you can run a in-depth website crawl using ACHE. Given a list of URLs (sites), ACHE will crawl all pages in each site. The crawler stops when no more links are found in the sites.

The process for running an in-depth crawl is simpler than for a focused crawl. An in-depth crawl doesn't require a model, it just needs a list of websites to crawl along with a configuration file with the appropriate settings for ACHE.

The following steps explain how to run such a crawl using ACHE.

1. **Prepare the seeds file** A seeds file should contain URLs of all websites that need to be crawled. A sample seeds file can be seen at config/sample.seeds.

2. **Create an `ache.yml` configuration file** The configuration file will contain the settings and rules the crawler will adhere to. A sample of the config file for running a in-depth website crawl is provided in config/config_website_crawl/ache.yml It contains the following important parameters:

   - `link_storage.link_strategy.use_scope` - Whether the crawler should crawl the websites provided as seeds only. This needs to be set to `true` in a in-depth website crawl.
   - `link_storage.download_sitemap_xml` - Whether to use the Sitemap protocol for discovery of links (if available). It helps to discover URLs more quickly.
   - `link_storage.scheduler.host_min_access_interval` - Configures the minimum time interval (in milliseconds) to wait between subsequent requests to the same host to avoid overloading servers. If you are crawling your own web site, you can descrease this value to speed-up the crawl.

3. **Run ACHE** Finally, when you have created the config file *ache.yml*, and the *seeds file*, you can run ACHE in the terminal:

   ```
   ache startCrawl -c <Path of Config Folder> -o <Path of Output Folder> -s
   ↪<Path of Seeds File>
   ```

   ACHE will continue crawling until interrupted using `CTRL+C` or until the queue of all links found during the crawl has been exhausted.

   For large crawls, you should consider using `nohup` for running the process in background:

```
nohup ache startCrawl -c <Path of Config Folder> -o <Path of Output Folder> -
↪s <Path of Seeds File> > crawler-log.txt &
```

CHAPTER 4

Running a In-Depth Website Crawl with Cookies

Some websites require users to login in order to access its content. ACHE allows crawling these type of websites by simulating the user login through sending the cookies along with HTTP requests.

The following steps show how to crawl sites that require login using ACHE.

1. **Get the cookies for each website using a web browser and also the user-agent string of that browser** The following instructions assume that you are using Chrome browser, although it should be similar with other browsers (i.e., Firefox and IE). For each website, repeat the following steps:

   - Sign up and login to the website.
   - Right click anywhere in the page and select `Inspect`. It will show the Developer Tools of the browser.
   - In the Developer Tools Bar on top, select the `Network` Tab.
   - Reload the page, to trigger the browser sending a request to the website. Then we will inspect this request to retrieve the cookie and user-agent string.
   - Select the first request in the `Name` panel on the left and a new panel with a tab named "Headers" will show up. Look for the section "Request Headers". Under this section, you need to locate and copy the values from the keys `cookie` and `user-agent` into ACHE's configuration file as shown in the next section.

2. **Create an `ache.yml` configuration file** The configuration file will contain the settings and rules the crawler will adhere to. A sample of the config file for running an in-depth website crawl with cookies is provided in config/config_login/ache.yml containing the following important parameters. Note that all the parameters are the same as ones used in in-depth website crawl except `crawler_manager.downloader.user_agent.string` and `crawler_manager.downloader.cookies`

   - `link_storage.link_strategy.use_scope` - Whether the crawler should crawl the websites provided as seeds only. This needs to be set to `true` in a in-depth website crawl.
   - `link_storage.download_sitemap_xml` - Whether to use the Sitemap protocol for discovery of links (if available). It helps to discover URLs quicker.

- `link_storage.scheduler.host_min_access_interval` - Configures the minimum time interval (in milliseconds) to wait between subsequent requests to the same host to avoid overloading servers. If you are crawling your own website, you can decrease this value to speed-up the crawl.
- `crawler_manager.downloader.user_agent.string` - The user-agent string acquired in the previous step.
- `crawler_manager.downloader.cookies` - A list of the website and its cookies acquired in the previous step. For example:

```
- domain: website1.com
  cookie: cookie1
- domain: website2.com
  cookie: cookie2
```

3. **Prepare the seeds file** A seeds file should contain URLs of all websites that need to be crawled. A sample seeds file can be seen at config/sample.seeds.

4. **Run ACHE** Finally, when you have created the config file *ache.yml*, and the *seeds file*, you can run ACHE in the terminal:

```
ache startCrawl -c <Path of Config Folder> -o <Path of Output Folder> -s
↪<Path of Seeds File>
```

ACHE will continue crawling until interrupted using `CTRL+C` or until the queue of all links found during the crawl has been exhausted.

For large crawls, you should consider using `nohup` for running the process in background:

```
nohup ache startCrawl -c <Path of Config Folder> -o <Path of Output Folder> -
↪s <Path of Seeds File> > crawler-log.txt &
```

CHAPTER 5

Crawling Dark Web Sites on the TOR network

TOR is a well known software that enables anonymous communications, and is becoming more popular due to the increasingly media on *dark web* sites. "Dark Web" sites are usually not crawled by generic crawlers because the web servers are hidden in the TOR network and require use of specific protocols for being accessed. Sites hidden on the TOR network are accessed via domain addresses under the top-level domain `.onion`. In order to crawl such sites, ACHE relies on external HTTP proxies, such as Privoxy, configured to route traffic trough the TOR network. Besides configuring the proxy, we just need to configure ACHE to route requests to `.onion` addresses via the TOR proxy.

Fully configuring a web proxy to route traffic through TOR is out-of-scope of this tutorial, so we will just use Docker to run the pre-configured docker image for Privoxy/TOR available at https://hub.docker.com/r/dperson/torproxy/. For convenience, we will also run ACHE and Elasticsearch using docker containers.

To start and stop the containers, we will use *docker-compose*, so make sure that the Docker version that you installed includes it. You can verify whether it is installed by running the following command on the Terminal (it should print the version of docker-compose to the output):

```
docker-compose -v
```

The following steps explain in details how to crawl `.onion` sites using ACHE.

1. Create the configuration files

All the configuration files needed are available in ACHE's repository at config/config_docker_tor (if you already cloned the git repository, you won't need to download them). Download the following files and put them in single directory named `config_docker_tor`:

1. tor.seeds: a plain text containing the URLs of the sites you want to crawl. In this example, the file contains a few URLs taken from https://thehiddenwiki.org/. If you want to crawl specific websites, you should list them on this file (one URL per line).
2. ache.yml: the configuration file for ACHE. It basically configures ACHE to run a in-depth website crawl of the seed URLs, to index crawled pages in the Elasticsearch container, and to download .onion addresses using the TOR proxy container.
3. docker-compose.yml: a configuration file for Docker, which specifies which containers should be used. It starts an Elasticsearch node, the TOR proxy, and ACHE crawler.

If you are using Mac or Linux, you can run the following commands on the Terminal to create a folder and download the files automatically:

```
mkdir config_docker_tor/
cd config_docker_tor/
curl -O https://raw.githubusercontent.com/ViDA-NYU/ache/master/config/config_
↪docker_tor/ache.yml
curl -O https://raw.githubusercontent.com/ViDA-NYU/ache/master/config/config_
↪docker_tor/docker-compose.yml
curl -O https://raw.githubusercontent.com/ViDA-NYU/ache/master/config/config_
↪docker_tor/tor.seeds
```

2. Start the Docker containers

Enter the directory `config_docker_tor` you just created and start the containers with docker-compose:

```
docker-compose up -d
```

This command will automatically download all docker images and start all necessary containers in background mode. The downloads may take a while to finish depending on your Internet connection speed.

3. Monitor the crawl progress

Once all docker images have been downloaded and the all services have been started, you will be able to open ACHE's web interface at http://localhost:8080 to see some crawl metrics. If you want to visualize the crawler logs, you can run:

```
docker-compose logs -f
```

4. Stop the Docker containers

You can stop the containers by hitting `CTRL+C` on Linux (or equivalent in your OS). You can also remove the containers by running the following command:

```
docker-compose down
```

Understanding the docker-compose.yml file

Basically, in `docker-compose.yml` we configure a container for the TOR proxy named `torproxy` that listens on the port 8118:

```
torproxy:
  image: dperson/torproxy
  ports:
    - "8118:8118"
```

An Elasticsearch node named `elasticsearch` that listens on the port 9200 (we also add some common Elasticsearch settings):

```
elasticsearch:
  image: elasticsearch:2.4.5
  environment:
    - xpack.security.enabled=false
    - cluster.name=docker-cluster
    - bootstrap.memory_lock=true
  ulimits:
    memlock:
      soft: -1
```

(continues on next page)

(continued from previous page)

```
    hard: -1
  volumes:
    - ./data-es/:/usr/share/elasticsearch/data # elasticsearch data will be␣
↪stored at ./data-es/
  ports:
    - 9200:9200
```

And finally, we configure a container named `ache`. Note that in order to make the config (`ache.yml`) and the seeds (`tor.seeds`) files available inside the container, we need to mount the volume `/config` to point to the current working directory. We also mount the volume `/data` in the directory `./data-ache` so that the crawled data is stored outside the container. In order to make ACHE communicate to the other containers, we need to link the ACHE's container to the other two containers `elasticsearch` and `torproxy`.

```
ache:
  image: vidanyu/ache
  entrypoint: sh -c 'sleep 10 && /ache/bin/ache startCrawl -c /config/ -s /config/tor.
↪seeds -o /data -e tor'
  ports:
    - "8080:8080"
  volumes:
    # mounts /config and /data directories to paths relative to path where this file␣
↪is located
    - ./data-ache/:/data
    - ./:/config
  links:
    - torproxy
    - elasticsearch
  depends_on:
    - torproxy
    - elasticsearch
```

Understanding the ache.yml file

The `ache.yml` file basically configures ACHE to index crawled data in the `elasticsearch` container:

```
# Configure both ELASTICSEARCH and FILES data formats, so data will be
# stored locally using FILES data format and will be sent to ELASTICSEARCH
target_storage.data_formats:
  - FILES
  - ELASTICSEARCH
# Configure Elasticsearch REST API address
target_storage.data_format.elasticsearch.rest.hosts:
  - http://elasticsearch:9200
```

and to download .onion addresses using the `torproxy` container:

```
crawler_manager.downloader.torproxy: http://torproxy:8118
```

All remaining configuration lines are regular ACHE configurations for running a in-depth website crawl of the seeds. Refer to the *in-depth website crawling turorial* for more details.

CHAPTER 6

Target Page Classifiers

ACHE uses target page classifiers to distinguish between relevant and irrelevant pages. Page classifiers are flexible and can be as simple as a simple regular expression, or a sophisticated machine-learning based classification model.

6.1 Configuring Page Classifiers

To configure a page classifier, you will need to create a new directory containing a file named `pageclassifier.yml` specifying the type of classifier that should be used and its parameters. ACHE contains several page classifier implementations available. The following subsections describe how to configure them:

- *title_regex*
- *url_regex*
- *body_regex*
- *regex*
- *smile* (a.k.a "weka" before version 0.11.0)

6.1.1 title_regex

Classifies a page as relevant if the HTML tag *title* matches a given pattern defined by a provided regular expression. You can provide this regular expression using the `pageclassifier.yml` file. Pages that match this expression are considered relevant. For example:

```
type: title_regex
parameters:
  regular_expression: ".*sometext.*"
```

6.1.2 url_regex

Classifies a page as relevant if the **URL** of the page matches any of the regular expression patterns provided. You can provide a list of regular expressions using the `pageclassifier.yml` file as follows:

```
type: url_regex
parameters:
  regular_expressions: [
    "https?://www\\.somedomain\\.com/forum/.*"
    ".*/thread/.*",
    ".*/archive/index.php/t.*",
  ]
```

6.1.3 body_regex

Classifies a page as relevant if the HTML content of the page matches any of the regular expression patterns provided. You can provide a list of regular expressions using the `pageclassifier.yml` file as follows:

```
type: body_regex
parameters:
  regular_expressions:
  - pattern1
  - pattern2
```

6.1.4 regex

Classifies a page as relevant by matching the lists of regular expressions provided against multiple fields: *url*, *title*, *content*, and *content_type*. You can provide a list of regular expressions for each of these fields, and also the type of boolean operation to combine the results:

- **AND** (default): All regular expressions must match
- **OR**: At least one regular expression must match

Besides the combination method for each regular expression within a list, you cab also specify how the final result for each field should be combined. The file `pageclassifier.yml` should be organized as follows:

```
type: regex
parameters:
    boolean_operator: AND|OR
    url:
      boolean_operator: AND|OR
      regexes:
        - pattern1-for-url
        - pattern2-for-url
    title:
      boolean_operator: AND|OR
      regexes:
        - pattern1-for-title
        - pattern2-for-title
    content:
      boolean_operator: AND|OR
      regexes:
        - pattern1-for-content
    content_type:
```

(continues on next page)

(continued from previous page)

```
    boolean_operator: AND|OR
    regexes:
      - pattern1-for-content-type
```

For example, in order to be classified as relevant using the following configuration, a page would have to:

- match regexes `.*category=1.*` OR `.*post\.php.*` in the URL
- AND
- it would have to match `.*bar.*` OR `.*foo.*` in the title.

```
type: regex
parameters:
    boolean_operator: "AND"
    url:
      boolean_operator: "OR"
      regexes:
        - .*category=1.*
        - .*post\.php.*
    title:
      boolean_operator: "OR"
      regexes:
        - .*bar.*
        - .*foo.*
```

6.1.5 smile (a.k.a "weka" before version 0.11.0)

Warning: This classifier was previously known as `weka` before version 0.11.0, and has been re-implemented using SMILE library which uses a more permissive open-source license (Apache 2.0). If you have models built using a previous ACHE version, you will need to re-build your model before upgrading ACHE to a version equal or greater than 0.11.0.

Classifies pages using a machine-learning based text classifier (SVM, Random Forest) trained using ACHE's `buildModel` command. Smile page classifiers can be built automatically by training a model using the command `ache buildModel`, as detailed in the next sub-section. You can also run `ache help buildModel` to see more options available.

Alternatively, you can use the Domain Discovery Tool (DDT) to gather training data and build automatically these files. DDT is an interactive web-based application that helps the user with the process of training a page classifier for ACHE.

A *smile* classifier supports the following parameters in the `pageclassifier.yml`:

- `features_file`, `model_file`: files containing the list of features used by the classifier and the serialized learned model respectively.
- `stopwords_file`: a file containing stop-words (words ignored) used during the training process;
- `relevance_threshold`: a number between 0.0 and 1.0 indicating the minimum relevance probability threshold for a page to be considered relevant. Higher values indicate that only pages which the classifier is highly confident are considered relevant.

Following is a sample `pageclassifier.yml` file for a smile classifier:

```
type: smile
parameters:
  features_file: pageclassifier.features
  model_file: pageclassifier.model
  stopwords_file: stoplist.txt
  relevance_threshold: 0.6
```

Building a model for the smile page classifier

To create the necessary configuration files, you will need to gather positive (relevant) and negative (irrelevant) examples of web pages to train the page classifier. You should store the HTML content of each web page in a plain text file. These files should be placed in two directories, named `positive`` and ``negative`, which reside in another empty directory. See an example at config/sample_training_data.

Here is how you build a model from these examples using ACHE's command line:

```
ache buildModel -t <training data path> -o <output path for model> -c <stopwords file␣
↪path>
```

where,

- `<training data path>` is the path to the directory containing positive and negative examples.
- `<output path>` is the new directory that you want to save the generated model that consists of two files: `pageclassifier.model` and `pageclassifier.features`.
- `<stopwords file path>` is a file with list of words that the classifier should ignore. You can see an example at config/sample_config/stoplist.txt.

Example of building a page classifier using our test data:

```
ache buildModel -c config/sample_config/stoplist.txt -o model_output -t config/sample_
↪training_data
```

6.2 Testing Page Classifiers

Once you have configured your classifier, you can verify whether it is working properly to classify a specific web page by running the following command:

```
ache run TargetClassifierTester --input-file {html-file} --model {model-config-
↪directory}
```

where,

- `{html-file}` is the path to a file containing the page's HTML content and
- `{model-config-directory}` is a path to the configuration directory containing your page classifier configuration.

CHAPTER 7

Crawling Strategies

ACHE has several configuration options to control the crawling strategy, i.e., which links the crawler should follow and priority of each link.

7.1 Scope

Scope refers to the ability of the crawler only follow links that point to the same "host". If the crawler is configured to use the "seed scope", it will only follow links that belong to the same host of the URLs included in the seeds file. You can enable scope adding the following line to `ache.yml`:

```
link_storage.link_strategy.use_scope: true
```

For example, if the scope is enabled and the seed file contains the following URLs:

```
http://pt.wikipedia.org/
http://en.wikipedia.org/
```

then the crawler will only follow links within the domains `pt.wikipedia.org` and `en.wikipedia.org`. Links to any other domains will be ignored.

7.2 Hard-focus vs. Soft-focus

The focus mode (hard vs. soft) is another way to prune the search space, i.e, discard links that will not lead to relevant pages. Relevant pages tend to cluster in connected components, therefore the crawler can ignore all links from irrelevant pages to reduce the amount of links that should be considered for crawling. In "hard-focus mode", the crawler will ignore all links from irrelevant pages. In "soft-focus mode", the crawler will not ignore links from irrelevant pages, and will rely solely on the link classifier to define which links should be followed and their priority. The hard-focus mode can be enabled (or disabled) using the following setting in `ache.yml`:

```
target_storage.hard_focus: true
```

When the hard focus mode is disabled, the number of discovered links will grow quickly, so the use of a link classifier (described bellow) is highly recommended to define the priority that links should be crawled.

7.3 Link Classifiers

The order in which pages are crawled depends on the `Link Classifier` used. A link classifier assigns a score (a double value) to each link discovered, and the crawler will crawl every link with a positive score with priority proportional to its score.

To configure link classifiers, you should add the key `link_storage.link_classifier.type` to `ache.yml` configuration file.

ACHE ships with several link classifier implementations, which are detailed next.

7.3.1 MaxDepthLinkClassifier

The max depth link classifier assigns scores to discovered links proportional to their depth the web tree (assuming the URLs provided as seeds are the roots) and will ignore any links whose depth is higher than the configured threshold.

For example, if you would like to crawl only the URLs provided as seeds, you could use the following configuration:

```
link_storage.link_classifier.type: MaxDepthLinkClassifier
link_storage.link_classifier.max_depth: 0
```

This configuration instructs ACHE to use the MaxDepthLinkClassifier and only crawl links within distance from the seeds equal to 0 (i.e., only the seeds).

If, instead, you use:

```
link_storage.link_classifier.type: MaxDepthLinkClassifier
link_storage.link_classifier.max_depth: 1
```

ACHE will crawl the seeds and every page linked from the seed pages (depth equals to 1).

Keep in mind that you can also combine this configuration with other configurations such as the scope. For example if you add the scope configuration as follows:

```
link_storage.link_strategy.use_scope: true
link_storage.link_classifier.type: MaxDepthLinkClassifier
link_storage.link_classifier.max_depth: 1
```

then ACHE will crawl only pages with a maximum depth of 1 AND belong to the sites provided as seeds. Without the scope configuration, ACHE would crawl pages from any web site with depth equal to 1.

7.3.2 LinkClassifierImpl

TODO

7.4 Online Learning

TODO

7.5 Backlink/Bipartite Crawling

TODO

CHAPTER 8

Data Formats

ACHE can store data in different data formats. The data format can be configured by changing the key `target_storage.data_format.type` in the configuration file.

The data formats currently available are:

- *FILESYSTEM_HTML, FILESYSTEM_JSON, FILESYSTEM_CBOR*
- *FILES*
- *WARC*
- *ELATICSEARCH*
- *KAFKA*

8.1 FILESYSTEM_*

Each page is stored in a single file, and files are organized in directories (one for each domain). The suffix in the data format name determines how content of each file is formatted:

- `FILESYSTEM_HTML` - only raw content (HTML, or binary data) is stored in files. Useful for testing and opening the files HTML using the browser.
- `FILESYSTEM_JSON` - raw content and some metadata is stored using JSON format in files.
- `FILESYSTEM_CBOR` - raw content and some metadata is stored using CBOR format in files.

When using any `FILESYSTEM_*` data format, you can enable compression (DEFLATE) of the data stored in the files enabling the following line in the configuration file:

```
target_storage.data_format.filesystem.compress_data: true
```

By default, the name of each file will be an encoded URL. Unfortunately, this can cause problems in some cases where the URL is very long. To fix this you can configure the file format to use a fixed size hash of the URL, instead of URL itself as a file name:

```
target_storage.data_format.filesystem.hash_file_name: true
```

Warning: All FILESYSTEM_* formats are not recommended for large crawls, since they can create millions files quickly and cause file system problems.

8.2 FILES

Raw content and metadata are stored in rolling compressed files of fixed size (256MB). Each file is a JSON lines file (each line contains one JSON object) compressed using the DEFLATE algorithm. Each JSON object has the following fields:

- `url` - The requested URL
- `redirected_url` - The URL of final redirection if it applies
- `content` - A Base64 encoded string containing the page content
- `content_type` - The mime-type returned in the HTTP response
- `response_headers` - An array containing the HTTP response headers
- `fetch_time` - A integer containing the time when the page was fetched (epoch)

8.3 WARC

Raw content and metadata are stored in WARC files. WARC is the standard format used by The Web Archive and other public web datasets such as "Common Crawl" and "ClueWeb". See http://commoncrawl.org/2014/04/navigating-the-warc-file-format/ for more details on the WARC format.

Every WARC file generated by ACHE contains one *warcinfo* entry and one *response* entry for each downloaded page. By default, the files are compressed using GZIP format and have an approximate size of 250MB (usually slightly larger). The default settings can be changed using the following entries in `ache.yml` file:

```
target_storage.data_format.type: WARC                     # enable WARC file format
target_storage.data_format.warc.compress: true            # enable GZIP compression
target_storage.data_format.warc.max_file_size: 262144000 # maximum file size in bytes
```

Finally, ACHE also stores additional metadata as non-standard extension WARC headers prefixed by `ACHE-*` (e.g., `ACHE-IsRelevant`, `ACHE-Relevance`).

8.4 ELASTICSEARCH

The ELASTICSEARCH data format stores raw content and metadata as documents in an Elasticsearch index.

8.4.1 Types and fields

Currently, ACHE indexes documents into one Elasticsearch type named `page` (or any name specified using the *command line* during the crawl initialization). The Elasticsearch mapping for this type is automatically created and contains the following fields:

- `domain` - domain of the URL
- `topPrivateDomain` - top private domain of the URL
- `url` - complete URL
- `title` - title of the page extracted from the HTML tag `<title>`
- `text` - clean text extract from HTML using Boilerpipe's DefaultExtractor
- `retrieved` - date when the time was fetched using ISO-8601 representation Ex: "2015-04-16T07:03:50.257+0000"
- `words` - array of strings with tokens extracted from the text content
- `wordsMeta` - array of strings with tokens extracted from tags `<meta>` of the HTML content
- `html` - raw HTML content
- `isRelevant` - indicates whether the page was classified as relevant or irrelevant by target page classifier. This is a keyword field (not analyzed string) containing either `relevant` or `irrelevant`.
- `relevance` - indicates the confidence of the target page classifier output. This is a decimal number with range from 0.0 to 1.0.

8.4.2 Configuration

To use Elasticsearch data format, you need to add the following line to the configuration file `ache.yml`:

```
target_storage.data_format.type: ELASTICSEARCH
```

You will also need to specify the host address and port where Elasticsearch is running. See the following subsections for more details.

REST Client (ACHE version >0.8)

Starting in version 0.8, ACHE uses the official Java REST client to connect to Elasticsearch. You can specify one or more Elasticsearch node addresses which the REST client should connect to using the following lines:

```
target_storage.data_format.elasticsearch.rest.hosts:
  - http://node1:9200
  - http://node2:9200
```

The following additional parameters can also be configured. Refer to the Elasticsearch REST Client documentation for more information on these parameters.

```
target_storage.data_format.elasticsearch.rest.connect_timeout: 30000
target_storage.data_format.elasticsearch.rest.socket_timeout: 30000
target_storage.data_format.elasticsearch.rest.max_retry_timeout_millis: 90000
```

Transport Client (deprecated)

You can also configure ACHE to connect to Elasticsearch v1.x using the native transport client by adding the following lines:

```
target_storage.data_format.elasticsearch.host: localhost
target_storage.data_format.elasticsearch.port: 9300
target_storage.data_format.elasticsearch.cluster_name: elasticsearch
```

8.4.3 Command line parameters

When running ACHE using Elasticsearch, you must provide the name of the Elasticsearch index that will be used as an argument to the CLI using the following parameters:

```
-e <arg>
```

or:

```
--elasticIndex <arg>
```

You can also (optional) provide the Elasticsearch type name to be used:

```
-t <arg>
```

or:

```
--elasticType <arg>
```

Run `ache help startCrawl` for more details on available parameters.

8.5 KAFKA

The KAFKA data format pushes crawled pages to an Apache Kafka topic. To configure this format, add the following lines to the `ache.yml` configuration file:

```
target_storage.data_format.type: KAFKA                    # enable KAFKA file format
target_storage.data_format.kafka.topic_name: mytopicname  # the name of the topic
target_storage.data_format.kafka.format: JSON             # value of messages will be␣
↪a JSON object
target_storage.data_format.kafka.properties:
  # The properties to be used while initializing the Kafka Producer
  bootstrap.servers: localhost:9092
  acks: all
  retries: 0
  batch.size: 5
  linger.ms: 100
  buffer.memory: 33554432
```

Currently, following message formats are supported:

- `JSON`: A JSON object using same schema defined in the *FILES* data format.
- `CDR31`: A JSON object formatted using the Memex CDR v3.1 format. Image objects are currently not supported.
- `ELASTIC`: A JSON object with the same fields described int the *ELATICSEARCH* data format.

CHAPTER 9

Link Filters

ACHE allows one to customize which domains and paths within a domain should be crawled. This can be done by configuring link filters using *regular expressions (regex)* or *wildcard* patterns. Regex filters are evaluated using Java's regular expression rules, and wildcard filters accept only the special character `*`, which matches any character.

Link Filters are composed of two lists of patterns:

- **whitelists** - patterns for URLs that are allowed to be followed, i.e., any URL that doesn't match the patterns is discarded.
- **blacklists** - patterns for URLs that are *NOT* allowed to be followed, i.e., any URL that matches the patterns is discarded.

Links filters can have **global** or **per-domain** scope. Global filters are evaluated against all links, whereas per-domain filters are evaluated only against URLs that belong to the specified domain (only top-private domain level). There are two ways to configure link filters:

- *.yml file*: Allows to configure global and per-domain link filters using YAML.
- *.txt files*: Allows to configure only regex-based global link filters.

9.1 Configuring using YAML

ACHE automatically searches for a file named `link_filters.yml` in the same directory of the `ache.yml` file. This file can contain a single `global` entry and one entry per domain. Each entry should specify a type (regex or wildcard) and a list of "whitelist" and "blacklist" patterns, as shown in the example bellow:

```
global:
  type: wildcard
  whitelist:
    - "http://*allowed*"
  blacklist:
    - "*bad-bad-pattern*"
www.example1.com:
  type: wildcard
```

(continues on next page)

(continued from previous page)

```
  blacklist:
    - http://www.example1.com/*disallowed*.html
  whitelist:
    - http://www.example1.com/*allowed*
www.example2.com:
  type: regex
  blacklist:
    - http:\/\/www\.example2\.com\/disallowed[0-9]+\.html
  whitelist:
    - http:\/\/www\.example2\.com\/allowed[0-9]+\.html
```

9.2 Configuring using .txt files

This is the old way to configure link filters. Only regex-based "global" filters can be configured, i.e., filters that are applied to all URLs. To configure a link filter, you will need to create text files containing one regular expression per line. All regular expressions loaded are evaluated against all links found on the web pages crawled in other to determine whether the crawler should accept or reject them. For whitelist filters, ACHE will automatically search for a file named `link_whitelist.txt`, whereas for blacklist filters the file name is `link_blacklist.txt`. These files should be placed under the same directory as the `ache.yml`. These files are loaded once during crawler start-up. The link filter files should look like this:

```
https?:\/\/www\.example\.com\/some_path\/.*
https?:\/\/www\.another-example\.com\/some_path\/.*
```

CHAPTER 10

Web Server & REST API

When an ACHE crawl is started, it automatically starts a REST API on port 8080. If that port is busy, it will try the following ports (8081, 8082, etc). The default HTTP settings can be changed using the following lines in the `ache.yml` file:

```
http.port: 8080
http.host: 127.0.0.1
http.cors.enabled: true
```

10.1 Security

There is only *HTTP Basic* authentication available at this time. To configure it, add the following lines to `ache.yml`:

```
http.auth.basic.user: myusername
http.auth.basic.password: mypasswd
```

10.2 Server Mode

Besides using the `ache startCrawl` command, ACHE can also be started in server mode and controlled using the web user interface or the REST API.

To start ACHE in server mode, you can use:

```
ache startServer -d /data -c /config/
```

Alternatively, if you are using Docker, run:

```
docker run -v $CONFIG:/config -v $DATA/data:/data vidanyu/ache startServer -d /data -
↪c /config/
```

where:

- `$CONFIG` is the path to where `ache.yml` is stored and
- `$DATA` is the path where ACHE is going to store its data.

If you want to configure a proxy to serve ACHE user interface from a **non-root** path, you will need to specify the path in `ache.yml` file using the following configuration:

```
http.base_path: /my-new-path
```

10.3 API Endpoints

POST /crawls/(string: *crawler_id*)/startCrawl

Starts a crawler with the crawler id `crawler_id`.

Request JSON Object

- **crawlType** (*string*) – Type of crawl to be started. Either `DeepCrawl` or `FocusedCrawl`.
- **model** (*string*) – (**Required for FocusedCrawl**) A base64 encoded string of the zipped model file. The zip file should contain the model files (`pageclassifier.yml`) and the seed file (`*_seeds.txt`).
- **seeds** (*array*) – (**Required for DeepCrawl**) An array of strings. Each string must be a fully-qualified URL that will be used starting point of the crawl.

Request body example for DeepCrawl:

```
{
  "crawlType": "DeepCrawl",
  "seeds": ["http://en.wikipedia.org/", "http://example.com/"],
  "model": null
}
```

Request body example for FocusedCrawl:

```
{
  "crawlType": "FocusedCrawl",
  "seeds": null,
  "model": "<Base64 encoded zipped model file>"
}
```

Response body example:

```
{
  "message": "Crawler started successfully.",
  "crawlerStarted": true
}
```

GET /crawls/(string: *crawler_id*)/status

Returns the status of the crawler with crawler id `crawler_id`.

Response body example:

```
{
  "status": 200,
  "version": "0.10.0",
```

(continues on next page)

(continued from previous page)

```
  "searchEnabled": false,
  "crawlerRunning": true,
  "crawlerState": "RUNNING"
}
```

GET /crawls/(string: *crawler_id*)/metrics

Returns detailed runtime metrics of the crawler with crawler id `crawler_id`. The metrics returned are generated using the *Dropwizard Metrics* library.

Response body example:

```
{
    "version": "3.1.3",
    "gauges": {
      "downloader.dispatch_queue.size": {
        "value": 0
      },
      "downloader.download_queue.size": {
        "value": 0
      },
      "downloader.pending_downloads": {
        "value": 2
      },
      "downloader.running_handlers": {
        "value": 1
      },
      "downloader.running_requests": {
        "value": 1
      },
      "frontier_manager.last_load.available": {
        "value": 0
      },
      "frontier_manager.last_load.rejected": {
        "value": 11610
      },
      "frontier_manager.last_load.uncrawled": {
        "value": 11610
      },
      "frontier_manager.scheduler.empty_domains": {
        "value": 0
      },
      "frontier_manager.scheduler.non_expired_domains": {
        "value": 1
      },
      "frontier_manager.scheduler.number_of_links": {
        "value": 2422
      },
      "target.storage.harvest.rate": {
        "value": 0.9777777777777777
      }
    },
    "counters": {
      "downloader.fetches.aborted": {
        "count": 0
      },
      "downloader.fetches.errors": {
        "count": 1
```

(continues on next page)

(continued from previous page)

```
      },
      "downloader.fetches.successes": {
        "count": 48
      },
      "downloader.http_response.status.2xx": {
        "count": 47
      },
      "downloader.http_response.status.401": {
        "count": 0
      },
      "downloader.http_response.status.403": {
        "count": 0
      },
      "downloader.http_response.status.404": {
        "count": 1
      },
      "downloader.http_response.status.5xx": {
        "count": 0
      },
      "target.storage.pages.downloaded": {
        "count": 45
      },
      "target.storage.pages.relevant": {
        "count": 44
      }
    },
    "histograms": {},
    "meters": {},
    "timers": {
      "downloader.fetch.time": {
        "count": 48,
        "max": 584.693196,
        "mean": 160.64529857175228,
        "min": 51.161457,
        "p50": 114.816344,
        "p75": 218.304927,
        "p95": 377.469511,
        "p98": 584.693196,
        "p99": 584.693196,
        "p999": 584.693196,
        "stddev": 118.74270199105285,
        "m15_rate": 0.4281665582051108,
        "m1_rate": 0.7030438799915493,
        "m5_rate": 0.4803778789487069,
        "mean_rate": 0.9178383293058442,
        "duration_units": "milliseconds",
        "rate_units": "calls/second"
      },
      [... Other metrics...]
    }
  }
```

GET /crawls/(string: *crawler_id*)/stopCrawl

Stops the crawler with crawler id `crawler_id` if it is running.

Query Parameters

- **awaitStopped** (*boolean*) – One of `true` or `false` (default). Indicates whether the request should block until the crawler is completely stopped.

Response body example:

```
{
  "message": "Crawler shutdown initiated.",
  "shutdownInitiated": true,
  "crawlerStopped": false
}
```

POST /crawls/(string: *crawler_id*)/seeds

Adds seeds to the crawler with crawler id `crawler_id`.

Request JSON Object

- **seeds** (*array*) – An array containing the URLs to be added to the crawl that is currently running.

Request body example:

```
{
  "seeds": ["http://en.wikipedia.org/", "http://example.com/"]
}
```

Response body example:

```
{
  "message": "Seeds added successfully.",
  "addedSeeds": true
}
```

CHAPTER 11

HTTP Fetchers

ACHE currently has multiple HTTP fetcher implementations based on different libraries (crawler-commons, which uses Apache HttpComponents, and okhttp3).

The fetchers support downloading URLs using both HTTP(S) and TOR protocol.

11.1 Switching the HTTP fetcher implementation

The default implementation uses Apache HttpComponents backend. To use the okhttp3 fetcher, you should enable it using:

```
crawler_manager.downloader.use_okhttp3_fetcher: true
```

11.2 Setting up a proxy

Crawling via a proxy is currently supported only by the okhttp3 fetcher. To configure this you can use the following lines:

```
# enable okhttp3 fetcher
crawler_manager.downloader.use_okhttp3_fetcher: true
# okhttp3 proxy configuration
crawler_manager.downloader.okhttp3.proxy_host: null
crawler_manager.downloader.okhttp3.proxy_username: null
crawler_manager.downloader.okhttp3.proxy_password: null
crawler_manager.downloader.okhttp3.proxy_port: 8080
```

11.3 Setting up a TOR proxy

In order to crawl links from the TOR network, an external TOR proxy is necessary. To set up such a proxy, you can use the following lines (note that only links from domains that end with *.onion* TLD use this proxy):

```
# Configure to download .onion URLs through the TOR proxy running at torproxy:8118
crawler_manager.downloader.torproxy: http://torproxy:8118
```

An example of crawling TOR network services is available at this *tutorial*.

CHAPTER 12

SeedFinder Tool

ACHE includes a tool called SeedFinder, which helps to discover more pages and web sites that contain relevant content. After you have your target page classifier ready, you can use SeedFinder to automatically discover a large set of seed URLs to start a crawl. You can feed SeedFinder with your page classifier and a initial search engine query, and SeedFinder will go ahead and automatically generate more queries that will potentially retrieve more relevant pages and issues them to a search engine until the max number of queries parameter is reached.

SeedFinder is available as an ACHE sub-command. For more instructions, you can run `ache help seedFinder`:

```
NAME
        ache seedFinder - Runs the SeedFinder tool

SYNOPSIS
        ache seedFinder [--csvPath <csvPath>] [(-h | --help)]
                --initialQuery <initialQuery> [--maxPages <maxPagesPerQuery>]
                [--maxQueries <maxNumberOfQueries>] [--minPrecision <minPrecision>]
                --modelPath <modelPath> [--searchEngine <searchEngine>]
                [--seedsPath <seedsPath>]

OPTIONS
        --csvPath <csvPath>
            The path where to write a CSV file with stats

        -h, --help
            Display help information

        --initialQuery <initialQuery>
            The inital query to issue to the search engine

        --maxPages <maxPagesPerQuery>
            Maximum number of pages per query

        --maxQueries <maxNumberOfQueries>
            Max number of generated queries

```

(continues on next page)

(continued from previous page)

```
        --minPrecision <minPrecision>
            Stops query pagination after precision drops bellow this minimum
            precision threshold

        --modelPath <modelPath>
            The path to the page classifier model

        --searchEngine <searchEngine>
            The search engine to be used

        --seedsPath <seedsPath>
            The path where the seeds generated should be saved
```

For more details on how SeedFinder works, you can refer to:

Karane Vieira, Luciano Barbosa, Altigran Soares da Silva, Juliana Freire, Edleno Moura. **Finding seeds to bootstrap focused crawlers.** World Wide Web. 2016. https://link.springer.com/article/10.1007/s11280-015-0331-7

CHAPTER 13

Frequently Asked Questions

13.1 What is inside the output directory?

Depending on the configuration settings, ACHE may creates different folders under the data output directory:

- **data_pages**: contains raw-data of crawled pages (including relevant, irrelevant and non-HTML content). The sub-directories and file formats depends on the configured *Data Format* being using. See *Data Formats* for more information.
- **data_monitor**: contains TSV-formatted log files with information about the status of the crawl, including relevant and irrelevant pages along with their scores, download requests and its metadata, etc.
- **data_url**, **data_backlinks**, **data_hosts**: are where the persistent storages keep data needed for crawler operation such as the frontier, the links graph, and metadata about crawled hosts.

13.2 When will the crawler stop?

The crawler will run until it downloads all links discovered during the crawling process, or until it hits maximum number of visited pages as configured in the `ache.yml` file. You can also look at `<data-output>/data_monitor/harvestinfo.csv` to check how many pages have been downloaded and decide whether you want to stop the crawler manually.

13.3 How to limit the number of visited pages?

By default, the maximum number of visited pages is set to the maximum integer value (*Integer.MAX_VALUE*). You can modify it by setting a value for the key `target_storage.visited_page_limit` in the `ache.yml` configuration file.

13.4 What format is used to store crawled data?

ACHE supports multiple types of the data formats. Take a look at the *Data Formats* page for more information. ACHE also supports indexing web pages directly into *Elasticsearch*.

13.5 How can I save irrelevant pages?

By default, this is off so you will need to set the value of `target_storage.store_negative_pages` to true in the configuration file.

13.6 Does ACHE crawl webpages in languages other than English?

ACHE does language detection and can be configured to ignore pages with non-English content. You can enable or disable language detection in the configuration file by changing `target_storage.english_language_detection_enabled`. Detection of other languages are currently not available, but could easily be supported in the future.

13.7 Is there any limit on number of crawled webpages per website?

There is no limit by default, but you can set a hard limit in the configuration file using the key `link_storage.max_pages_per_domain`. You can enable this so that the crawler doesn't get trapped by particular domains, and to favor crawling a larger number of domains as opposed to focusing on a few domains.

13.8 Why am I getting a *SSL Handshake Exception* for some sites?

A `javax.net.ssl.SSLHandshakeException : handshake_failure` usually occurs when the server and ache can't decide on which Cipher to use. This most probably happens when the JVM is using a limited security cipher suite. The easiest work around for this is to use OpenJDK 8+ because it comes with Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy out of the box. To install this JCE on Oracle, follow the instructions here.

13.9 Why am I getting a *SSL Protocol Exception* for some sites?

A `javax.net.ssl.SSLProtocolException : unrecognized_name` is a server misconfiguration issue. Most probably, this website is hosted on a virtual server. A simple solution is to disable SNI extension by adding `-Djsse.enableSNIExtension=false` as VM options when running ACHE. However, keep in mind that disabling SNI will cause certificate validation failures for some sites which use mutiple host-names behind a shared IP.

13.10 Where to report bugs?

We welcome feedback. Please submit any suggestions or bug reports using the GitHub issue tracker (https://github.com/ViDA-NYU/ache/issues)

CHAPTER 14

Links

- GitHub repository

HTTP Routing Table

/crawls

GET /crawls/(string:crawler_id)/metrics, 31
GET /crawls/(string:crawler_id)/status, 30
GET /crawls/(string:crawler_id)/stopCrawl, 32
POST /crawls/(string:crawler_id)/seeds, 33
POST /crawls/(string:crawler_id)/startCrawl, 30