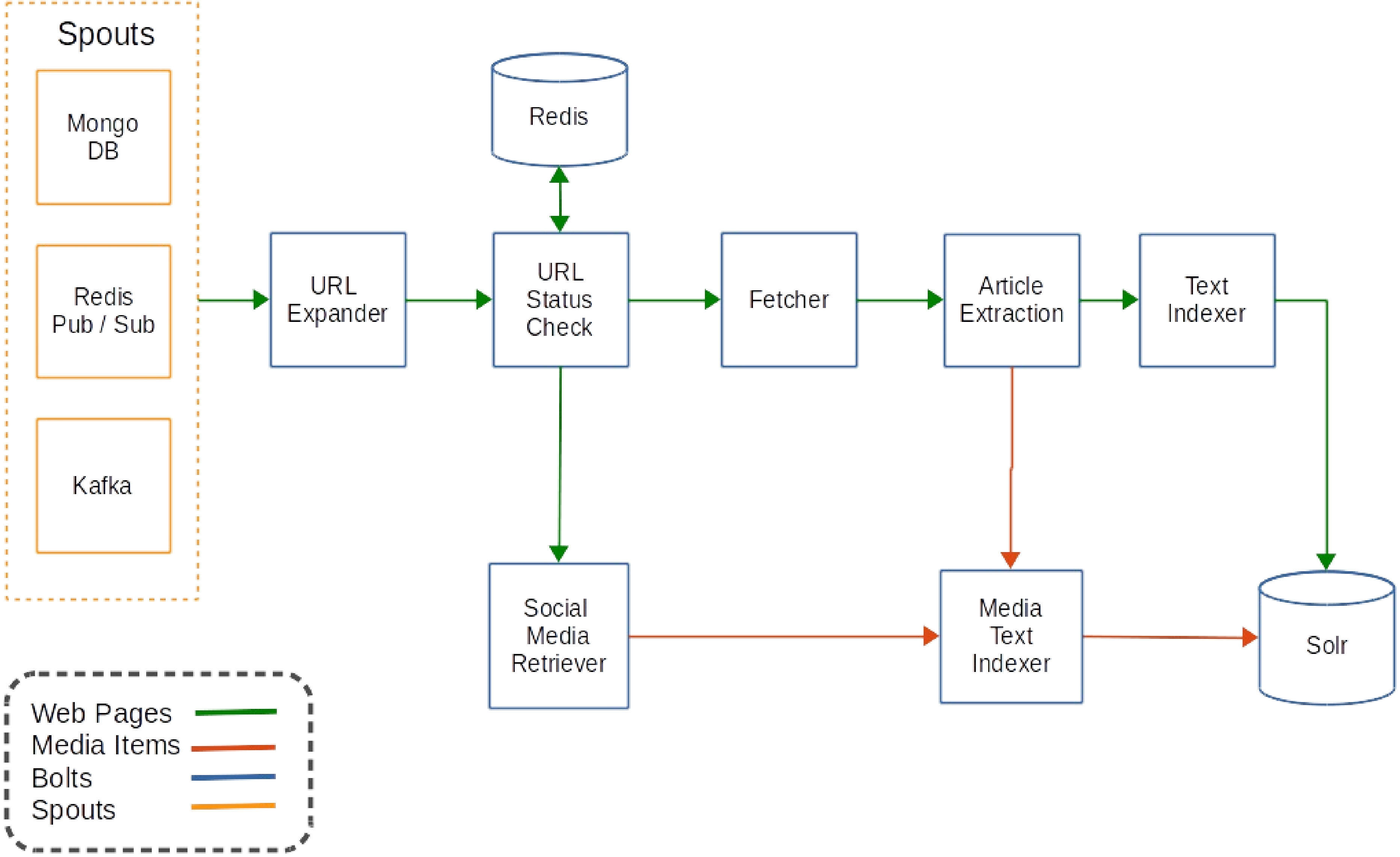
mklab-focused-crawler

The main purpose of mklab-focused-crawler is fetching, parsing, analysis and indexing of web pages shared through social networks. Also this module collects multimedia content embedded in webpages or shared in social media platforms and index it for future use.

The main pipeline of focused crawler is implemented as a storm topology, where the sequential bolts perform a specific operation on the crawling procedure. The overall topology of focused crawler is depiced in the following figure.


Spouts
Mongo DB
Redis Pub / Sub
Kafka
URL Expander
Redis
URL Status Check
Fetcher
Article Extraction
Text Indexer
Social Media Retriever
Media Text Indexer
Solr
Web Pages
Media Items
Bolts
Spouts

The input stream consists of URLs: these refer either to arbitrary web pages or to social media pages. There are three spouts that are possible to inject URLs in the topology: a) one that periodically reads URLs from a running mongo database instance, b) one that listens to a Redis message broker following the Publish/Subscribe pattern, and c) one waiting for URLs from a Kafka queue. The URLs fed to the crawler may be produced by any independent process. One possibility is to use the Stream Manager project.

The web pages injected in the topology have the following json structure:

```
{
    "_id": "https://youtu.be/zvad7iztAIM",
    "url": "https://youtu.be/zvad7iztAIM",
    "date": ISODate("2016-07-11T10:37:17.0Z"),
    "reference": "Twitter#752451689530089473",
    "source": "Twitter"
}
```

There is a url (usually shortened), a publication date, the id of the social media item contain that url, and an identifier of the social media platform. While web page objects pass through the topology, they are updated with additional fields.

The first bolt in the topology deserializes the messages injected in the topology by the spouts, from a json object to WebPage objects. As URLs on Twitter are usually shortened, the next bolt (URLExpansionBolt) expands them to long form. The expanded urls is added in the corresponding field in the WebPage object.

The next bolt checks the type of the URLs and its crawling status. URLs that correspond to posts in popular social media platforms (e.g., https://www.youtube.com/watch?v=LHAZYK6x6iE) are redirected to a bolt named MediaExtractionBolt, which retrieves metadata from the respective platforms.

URLs to arbitrary web pages are emitted to a Fetcher bolt. Non-HTML content is discarded. The fetched content is then forwarded to the next bolt (ArticleExtractionBolt) that attempts to extract articles and embedded media items. In case of articles the title and the main text of the article is extracted and added as additional fields in the WebPage objects.

The extracted articles are indexed in a running Solr instance by the Text Indexer. The extracted media items, as well as the media items coming from the MediaExtractionBolt are handled by the Media Text Indexer.

Building & Configuration

To run the predefined topology (with Redis as a spout) a set of parameters has to be specified in the configuration file. The first parameters have to be specified are those concern the running instance of redis:

```
    <redis>
        <hostname>xxx.xxx.xxx.xxx</hostname>
        <port>6379</port>
        <webPagesChannel>DiceWebPages</webPagesChannel>
    </redis>
```

The next part is the section that specifies solr parametets:

```
  <textindex>
      <host>xxx.xxx.xxx.xxx</host>
      <port>8983</port>port>
      <collections>
           <webpages>WebPages</webpages>
           <media>MediaItems</media>
      </collections>
  </textindex>
```

The running instance of Solr has to contain to cores, corresponding to web pages and media items. The configuration files and the schema of each of these cores can be found here.

To activate media extraction a set of credentials has to be specified for YouTube, Facebook, Instagram and Flickr. The corresponding section is the following:

```
<mediaextractor>
  <youtube>
    <key></key>
  </youtube>
  <facebook>
    <accesstoken></accesstoken>
  </facebook>
```

```
    <instagram>
      <accesstoken></accesstoken>
      <accesstokensecret></accesstokensecret>
    </instagram>
    <flickr>
      <key></key>
      <secret></secret>
    </<flickr>>
  </mediaextractor>
```

Finally, to specify whether storm topology will run in a storm cluster or in local mode the following section has to be set:

```
<topology>
    <focusedCrawlerName>DiceFocusedCrawler</focusedCrawlerName>
    <local>true</local>
</topology>
```

If *local* parameter is true then the topology will run in a simulated local cluster:

```
      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology(name, conf, topology);
```

In other case the topology will be submitted in a running storm cluster:

```
    StormSubmitter.submitTopology(name, conf, topology);
```

Note that setting of parameters in the configuration file, must be performed before jar building, as that jar has to contain all the necessary files for execution. To build the executable jar use the following mvn command:

```
$mvn clean assembly:assembly
```

The generated jar, named *focused-crawler-jar-with-dependencies.jar*, contains all the dependencies and the configuration file described above and can be used for submission in a running storm cluster. The main class of the topology is *gr.iti.mklab.focused.crawler.DICECrawler*. This entry point is specified in the pom.xml file in the maven-assembly-plugin.

To submit on storm:

```
storm jar focused-crawler-jar-with-dependencies.jar
                      gr.iti.mklab.focused.crawler.DICECrawler
```

The following parameters of storm can be specified in the topology level in order to maximize the performance of crawler:

- Config.TOPOLOGY_WORKERS: This sets the number of worker processes to use to execute the topology. For example, if you set this to 25, there will be 25 Java processes across the cluster executing all the tasks. If you had a combined 150 parallelism across all components in the topology, each worker process will have 6 tasks running within it as threads.

- Config.TOPOLOGY_MAX_SPOUT_PENDING: This sets the maximum number of spout tuples that can be pending on a single spout task at once (pending means the tuple has not been acked or failed yet). It is highly recommended you set this config to prevent queue explosion.

- Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS: This is the maximum amount of time a spout tuple has to be fully completed before it is considered failed. This value defaults to 30 seconds, which is sufficient for most topologies. See Guaranteeing message processing for more information on how Storm's reliability model works.