BFS with MapReduce and Wikipedia Crawler

Nyaika David
University of Trento
MAT.174258
davie2086@gmail.com

ABSTRACT

Actual paper is a **"Big Data"** course project at **University of Trento, Italy**.It mainly targeted at building a **web crawler** with a single purpose: data-mine the entire wikipedia onto a single laptop using parallel computation and NoSQL database. Paper contains references to the source code freely available on **github** accounts as well as instructions on how to run them and what technologies have been used.

1. PART B: TECHNOLOGICAL

The technological section focuses on crawling large amounts of web pages from Wikipedia as a data source. HTML pages are downloaded in parallel using spark. Data filtering is later on performed on the downloaded data converting them into strings which are persisted into mongodb.

1.1 Implementation

The implementation phase is two-fold, it consists of a crawler which crawls HTML pages from Wikipedia and an implementation in spark which runs the crawler in parallel, saving the crawled HTML pages as strings into mongodb.

The crawler code is taken from an implementation that is packaged with **crawler4J** [1] a java API. Spark implementation is built using **spark java API (v1.4.0)**,**maven** for adding code dependencies to the class path and, **netbeans(v8.0.2)** integrated development environment for the development and **mongodb(v3.0.2)** for storage. The entire technological section is implemented using 4 classes and test file for storing seed URLs:

- **WebCrawlerImpl** This class decides which URLs should be crawled and handles the downloaded page [1]
- **CrawlControllerImpl** This class specifies the seed URLs of the crawl, the folder in which intermediate crawl data should be stored and the number of concurrent threads [1]
- **Pages** This class represents a downloaded page
- **SparkImpl** This class stores the Map Reduce implementation for spark
- **seedurls** This is a text file that stores the seed URLs separated by commas

Below is a sample implementation of the WebCrawlerImpl class [1]. In this there are two main functions shouldVisit: This function decides whether the given URL should be crawled or not. In the above code snippet, the does not allow .css, .js and media files and only allows pages within 'https://en.wikipedia.org/wiki/' domain. visit: This function is called after the content of a URL is downloaded successfully. You can easily get the URL, text, links, html, and unique id of the downloaded page.

```
private final static Pattern FILTERS =
     Pattern.compile(".*\\.(bmp|gif|jpg|png)$");

   /**
    * This method receives two parameters. The first
        parameter is the page
    * in which we have discovered this new url and the
        second parameter is
    * the new url. You should implement this function
        to specify whether
    * the given url should be crawled or not (based on
        your crawling logic).
    * In this example, we are instructing the crawler
        to ignore urls that
    * have css, js, git, ... extensions and to only
        accept urls that start
    * with "http://www.ics.uci.edu/". In this case, we
        didn't need the
    * referringPage parameter to make the decision.
    */
    @Override
    public boolean shouldVisit(Page referringPage,
        WebURL url) {
       String href = url.getURL().toLowerCase();
       return !FILTERS.matcher(href).matches()
              &&
                   href.startsWith("https://en.wikipedia.org/wiki/");
    }

    /**
     * This function is called when a page is fetched
         and ready
     * to be processed by your program.
     */
    @Override
    public void visit(Page page) {
       String url = page.getWebURL().getURL();
```

```
        System.out.println("URL: " + url);

        if (page.getParseData() instanceof
             HtmlParseData) {
            HtmlParseData htmlParseData =
                 (HtmlParseData) page.getParseData();
            String text = htmlParseData.getText();
            String html = htmlParseData.getHtml();
            Set<WebURL> links =
                 htmlParseData.getOutgoingUrls();

            System.out.println("Text length: " +
                 text.length());
            System.out.println("Html length: " +
                 html.length());
            System.out.println("Number of outgoing
                 links: " + links.size());
        }
    }
```

Below is the sample code of CrawlControllerImpl class [1]. This class specifies the seed URLs of the crawl, the folder in which intermediate crawl data should be stored and the number of concurrent threads:

```
public static void executeController(String seedUrl)
     throws Exception {
       String crawlStorageFolder = "/data/crawl/root";
       int numberOfCrawlers = 3;

       CrawlConfig config = new CrawlConfig();
       config.setCrawlStorageFolder(crawlStorageFolder);

       /*
        * Instantiate the controller for this crawl.
        */
       PageFetcher pageFetcher = new
            PageFetcher(config);
       RobotstxtConfig robotstxtConfig = new
            RobotstxtConfig();
       RobotstxtServer robotstxtServer = new
            RobotstxtServer(robotstxtConfig,
            pageFetcher);
       CrawlController controller = new
            CrawlController(config, pageFetcher,
            robotstxtServer);

       /*
        * For each crawl, you need to add some seed
             urls. These are the first
        * URLs that are fetched and then the crawler
             starts following links
        * which are found in these pages
        */
  controller.addSeed(seedUrl);

    /*
        * Start the crawl. This is a blocking
             operation, meaning that your code
        * will reach the line after this only when
             crawling is finished.
        */
controller.start(WebCrawlerImpl.class,
     numberOfCrawlers);
```

Below is the SparkImpl class. This class contains the MapReduce implementation. The flatMapToPair(..) method maps a URL to its corresponding page and returns JavaPairRDD that contains a list of URL-page pairs.

The reduceByKey(...) method creates a list of pairs with similar keys which are later on persisted in mongodb document as a key and value.

```
public static void main(String[] args) throws
     Exception {
       SparkConf sparkConf = new
            SparkConf().setAppName("WebCrawler").setMaster("local");
       JavaSparkContext ctx = new
            JavaSparkContext(sparkConf);
       JavaRDD<String> lines = ctx.textFile("seedurls",
            1);

       JavaRDD<String> urls = lines.flatMap(new
            FlatMapFunction<String, String>() {
           @Override
           public Iterable<String> call(String s) {
              return Arrays.asList(COMMA.split(s));
           }
       });

       // Mapping URLs and HTMLs for reducer
       JavaPairRDD<String, Page> map =
            urls.flatMapToPair(new
            PairFlatMapFunction<String, String, Page>()
            {
           @Override
           public Iterable<Tuple2<String, Page>>
                call(String string) throws Exception {
              CrawlControllerImpl.executeController(string);
              Set<Tuple2<String, Page>> crawledData =
                   new
                   HashSet<>(CrawlControllerImpl.crawledData.size());

              for (Map.Entry<String, String> pair :
                   CrawlControllerImpl.crawledData.entrySet())
                   {
                  crawledData.add(new
                       Tuple2<>(pair.getKey(), new
                       Page(pair.getKey(),
                       pair.getValue())));
              }
              logger.info("map:::" + crawledData);

              return crawledData;
           }
       });

       // Saving HTML and URL as key to MongoDB
       JavaPairRDD<String, Page> reducer =
            map.reduceByKey(new Function2<Page, Page,
            Page>() {
           @Override
           public Page call(Page page1, Page page2) {
              logger.info("here iam dummy!!!!");
              Page page = Page.mergePages(page1, page2);
              new
                   MongoClient().getDatabase("bigDCourse").getCollecti
                     new Document("webpage",
                            new Document().append(
                                   page.getUrl(),
                                        page.getHtml())));
              return page;
           }
       });
```

1.2 Challenges

Crawling too much data throws out of memory error. Data structures for MapReduce do not better support the data crawling task. Having a single element in the mapper will cause reducer step being skipped. That is happening because reduceByKey method expects at least two elements,

but getting only one.

1.3 Conclusions

Spark provides an advantage of efficiency in terms of speed towards the crawling job given the fact that it is executed in parallel. MapReduce has been used in performing the task however, the data structures such as RDDs provided by the MapReduce framework provide no value towards carrying out the task assigned.Furthermore, MapReduce framework better supports computation problems such as addition

2. REFERENCES

[1] Y. Ganjisaffar. crawler4j. `http://https://github.com/yasserg/crawler4j`, 2015. [Online; accessed 15-July-2015].