Hadoop MapReduce in Nutch

DAVIDE SCANO- COMPUTER SCIENCE & NETWORKING - 457926

1 INTRODUCTION

Nutch is a well matured, production ready Web crawler and it enable fine grained configuration, relying on Apache Hadoop data structures. Being pluggable and modular of course has it's benefits, Nutch provides extensible interfaces such as Parse, Index and ScoringFilter's for custom implementations. Nutch can find Web page hyperlinks in an automated manner, reduce lots of maintenance work, for example checking broken links, and create a copy of all the visited pages for searching over.

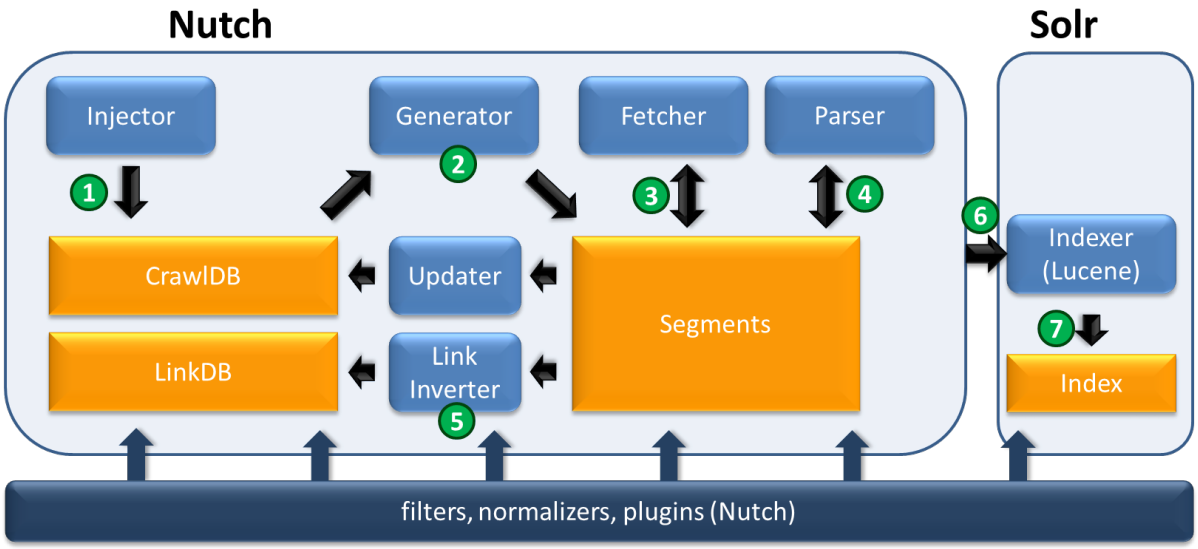

Fig. 1. Nutch architecture[1]

1) The injector takes all the URLs of the seeds.txt file and adds them to the crawldb. As a central part of Nutch, the crawldb maintains information on all known URLs (fetch schedule, fetch status, metadata, ...).
2) Based on the data of crawldb, the generator creates a fetchlist and places it in a newly created segment directory.
3) Next, the fetcher gets the content of the URLs on the fetchlist and writes it back to the segment directory.

4) Now the parser processes the content of each web page. If the crawl functions as an update or an extension to an already existing one (e.g. depth of 3), the updater would add the new data to the crawldb as a next step.
5) Before indexing, all the links need to be inverted, which takes into account that not the number of outgoing links of a web page is of interest, but rather the number of inbound links, they are saved in the linkdb.
6) Using data from all possible sources (crawldb, linkdb and segments), the indexer creates an index and saves it within the Solr directory. For indexing, the popular Lucene library is used. Now, the user can search for information regarding the crawled web pages via Solr.
7) Filters, normalizers and plugins allow Nutch to be highly modular, flexible and very customizable throughout the whole process.

Nutch 1.x installation is pretty simple, is necessary just follow the guide at this page https://wiki.apache.org/nutch/NutchTutorial. Acheiving as our main goal is understand how Nutch modules work, we have to see the source code of they are made and for achieve this purpose we use Eclipse i.e IDE. The Nutch documentation provide a useful guide for set up correctly the environment https://wiki.apache.org/nutch/RunNutchInEclipse. In this report for each chapters we will present how to set up or initialize a certain module and after that will focus on the algorithm based on MapReduce programming model supplied by Hadoop framework that is used in that Nutch module for doing his job. Algorithms will be described using pseudocode and tables will be used for represent the data that they have to manage.

1.1 CrawlDatum

CrawlDatum objects contains the crawl state of URL, they are the Nutch backbone because they are used in many modules of these architecture. The class CrawlDatum is defined as *implements* and it implements two interfaces: *WritableComparable<CrawlDatum>* and *Cloneable*. A *WritableComparable* is *Writable* which is also *Comparable* i.e objects can be compared to each other, typically via *Comparator*s. A class implements the *Cloneable* interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class. At the beginning of the class there are two HasMap: *oldToNew*<Byte,Byte>,*statNames*<Byte,String>. The first HasMap maintains the byte value related to status transaction of url for example the url pass from STATUS_DB_UNFETCHED = 0x01 to STATUS_DB_FETCHED = 0x02. The second HashMap maintains the relation between the byte value of the state and is related string name for example <0x01 , "db_unfetched">. This two structure are suitable for maintain the status information when url is manage by modules of the architecture. There are some other important information carried by CrawlDatum such as fetch time that tell us when url was, fetch intervals i.e a refresh interval, score that express an URL preference, Metadata that store keys reserved for setting a custom score for a specific URL, etc. Additionally the class CrawlDatum have various methods suitable for sets, gets and

Crawl Information	
Version: 7 Status: 1 (db_unfetched) Fetch time: Sun Mar 10 11:25:54 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000seconds (30 days) Score: 7.0 Signature: null Metadata:	Version: 7 Status: 1 (db_unfetched) Fetch time: Sun Mar 10 11:25:54 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000seconds (30 days) Score: 1.0 Signature: null Metadata:

Table 1. CrawlDatum

manage the informations carried by the object, for sake of clarify we can see and example of CrawlDatum object in *Table1*.

2 INJECTOR

2.1 Setup and testing

Injector is the first module of Nutch crawler and he creates/initializes CrawlDb that stores the current status of each URL, as a map file of <url, CrawlDatum>, where keys use Text and values use CrawlDatum object. The Injector module simply converts plain-text representation of the initial list of URLs, called the seed list showed *Table2*, to a map file and subsequently CrawlDb is updated with the information from the fetched and parsed pages.

Seeds file
https://www.unipi.it #https://www.UNIPI.it https://www.amazon.it nutch.score=10 https://www.google.it nutch.score=7 https://www.ansa.it

Table 2. Seeds list

In *Table2* some urls are associated with the "command"**nutch.score**, that is use to assign a certain score value at the url suitable for the next operations. For execute "injection module" in Eclipse we have to **org.apache.nutch.crawl** package and find the **Injector** java class, after that you have to configure the "run configuration". For run the injector we use: *bin/nutch inject inject/crawldb urls/seeds.txt*
For sake of clarify we specify better the program arguments, that are:

- **CrawlDb**: The directory containing the crawldb i.e the output file.
- **Url_dir**:The directory containing our seed list (referred to above as 'flat file'), usually a text document containing URLs, one URL per line, i.e the input file.

The VM argumets is configured for storing the log information, will be useful for understand/see how the module phases proceed. For better understand how the class works, we can obtain some sneak peek of the processing data using java **LOG** that deal with printing in log and Eclipse console the selected "data".

2.2 MapReduce algorithm

The class *InjectMapper*, at the beginning, initialize some variables using some informations carried by the context for example: filters, normalizers, etc. In the pseudocode we can see the structure of *Map* method, that takes in input URL *u*,from seeds file, and CrawlDatum *d* that contain crawl state/information of the associated url.

Algorithm 1
Inject Mapper

```
1: class InjectMapper
2:     method Map(Url u, CrawlDatum d)
3:         url ← filterNormalize(u)
4:         datum ← new crawlDatum                ▷ For storing injected information
5:         datum ← datum.getInfo(d)
6:         if metadata.length() > 0 then
7:             processMetaData (metadata, datum, url)    ▷ Extract metadata from the url
8:         else
9:             EMIT(url, datum)
```

First of all in the *Map* method we apply *filterNormalize* that is in charges to filter and normalize *u* according to how we have wrote the filter in configuration files, so we obtain *url*, in *Table 1* we can see some example. After that we create *datum* that is an object CrawlDatum, where we will store the crawl state of the *url*, if we have some metadata liked with *Url* we use *processMetaData* for extract and update the suitable parameter in *datum*. In the *Table 4* we can see the couple outputted by Map that will go to the Reduce.

Url seeds	Metadata	Normalize Url
https://www.Google.it nutch.score=7	nutch.score=7	https://www.google.it
#https://www.UNIPI.it	\	https://www.unipi.it
https://www.amazon.it nutch.score=10	nutch.score=10	https://www.amazon.it

Table 3. Url manipulation

url	datum
https://www.unipi.it/t	Version: 7 Status: 66 (injected) Fetch time: Sun Mar 10 11:25:54 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000seconds (30 days) Score: 1.0 Signature: null Metadata:
https://www.amazon.it/	Version: 7 Status: 66 (injected) Fetch time: Sun Mar 10 11:25:54 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000seconds (30 days) Score: 10.0 Signature: null Metadata:

Table 4. Map outputed couple

At the beginning of the class *InjectReducer* we define three objects CrawlDatum: *inject*, *old*, *result*; This variables are suitable for store the crawl state informations related to a certain URL, soon we will see more in detail their "job".
The main goal for the Reducer is combine multiple new entries for a URL using this rules:

Algorithm 2 Inject Reducer

```
1:  class InjectReducer
2:      inject ← new CrawlDatum
3:      old ← new CrawlDatum
4:      result ← new CrawlDatum                    ▷ Store reduce injected information
5:      method Reduce(Url u, datum[d1, d2, ..])
6:          oldSet ← false
7:          injectedSet ← false
8:          for values ∈ datum[d1, d2, ..] do
9:              if values.getStatus() == "injected" then
10:                 inject.set(values)
11:                 injectedSet ← true
12:             else
13:                 old.set(values)
14:                 oldSet ← true
15:         if injectedSet then
16:             result ← inject
17:         if oldSet then
18:             result ← old
19:         EMIT(u, result)
```

- If there is only new injected record we emit injected record.
- If there is only old record we emit existing record.
- If both new and old records are present, we can chose if overwrite and emit injected record or overwrite and update and we update existing record and emit it.

The *Reducer* method takes in input the output (*u* and *datum*) emitted by the *Map* sorted by key, thanks to shuffle and sort performed by Hadoop between Mapper and Reducer. In the for cycle we iterate over the *values* contained in the iterable object *datum*. We check if the status of *values* is *"injected"*, if is it true, we copy the value in *inject* otherwise copies the value in *old*. In the real world scenario the *if* where we check *values* status is more articulated because he have to deal with the "control" presented before in the bulleted list. Let me clarify the last sentence better, in the CrawlDb folder there are two subfolders, current and old, where we store information for the first time crawled url and that I've already been crawled. If we inject for the first time a certain url, the Mapper produce just one CrawlDatum object associate to the url with *Status: 66 (injected)* and when we move into the Reducer *inject* become true and in the *result* we store the information contained in *inject*. Now if we crowded another time the same url, the Mapper at this time produce two CrawlDatum one with *Status: 1 (db_unfetched)* and one *Status: 66 (injected)* so in this case in the Reducer *inject* become true and even *old*

become true, now when we move into Reducer the if in row 9 have to handle the third point of bulleted list. After that depending on which boolean variable is *true* we store the data in *result* and at the end reducer produce a couple formed by *u*, *result*. In the first column of *Table 5* we have input urls whereas second and third column we can see the output couple produced by the reducer, it is noticeable that repeat input urls are merge in a single output couple.

Url input	u	result
https://www.google.it nutch.score=7	https://www.google.it	Version: 7 Status: 1 (db_unfetched) Fetch time: Sun Mar 10 11:25:54 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000seconds (30 days) Score: 7.0 Signature: null Metadata:
https://www.unipi.it #https://www.UNIPI.it https://www.UnIpI.it	https://www.unipi.it	Version: 7 Status: 1 (db_unfetched) Fetch time: Sun Mar 10 11:25:54 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000seconds (30 days) Score: 1.0 Signature: null Metadata:

Table 5. Crawl information related to Url in seeds file

3 GENERATOR

3.1 Setup and testing

Generator is the second module of Nutch crawler, into the *segments* folders he create the *crawl_generate* that contains the list of URLs to be fetched, together with their current status retrieved from CrawlDb, as a sequence file of <url, CrawlDatum>. This data uses sequence file, first because it's processed sequentially, and second because we couldn't satisfy the map file invariants of sorted keys. We need to spread URLs that belong to the same host as far

apart as possible to minimize the load per target host, and this means that records are sorted more or less randomly. In Eclipse we have to **org.apache.nutch.crawl** package and find the **Generator** java class, after that you have to configure the "run configuration". For run the generator we use: *bin/nutch generate /inject/crawldb segments -topN 2 -numFetchers 2*
For better clear up will present all the possible arguments that we can set up:

- **CrawlDb**: Path to the location of our CrawlDb directory
- **segments_dir**: Path to the location of our segments directory where the Fetcher Segments are created.
- **force**: This argument will force an update even if there appears to be a lock.
- **topN N**: Where N is the number of top URLs to be selected
- **numFetchers numFetchers**: The number of fetch partitions. Default: Configuration key -> mapred.map.tasks -> 1 (in local mode), possibly multiple in deploy/distributed mode.
- **adddays numDays**: Adds <days> to the current time to facilitate crawling urls already fetched sooner then db.default.fetch.interval. Default: 0.
- **noFilter**:Whether to filter URLs or not is read from the crawl.generate.filter property in nutch-site.xml/nutch-default.xml configuration files. If the property is not found, the URLs are filtered.
- **noNorm**: The exact same applies for normalization parameter as does for the filtering option above.
- **maxNumSegments num**: The (maximum) number of segments to be generated. Default: 1 – Note: if multiple segments are generated, the limit -topN applies to the total number of URLs for all segments taken together, while generate.max.count is applied to every generated segment individually.

Using well configured VM arguments and java LOG in Eclipse will be useful for understand how does it work the module.

3.2 MapReduce algorithm

At the beginning of *SelectorMapper* we have to define/initialize some objects, using the data contains in the context. These object are in charge to filtering, scoring and checking the input parameters in *Map* method. The *Map* takes in input an URL *u* and CrawlDatum *d*, the first *if* check if there are present some URL filters to apply on *u*, for example one of this filter could be obtained configuring *regex-urlfilter.txt* with a regular expression matching the domain you wish to crawl. In the second *if*, method *shouldFetch* takes in input *u*, *d* ,curTime and this last parameter is a reference time and usually is set to the time when the fetchlist generation process was started.
shouldFetch provides information whether the page is suitable for selection in the current fetchlist and it return a boolean and if is true does not guarantee that the page will be fetched, it just allows it to be included in the further selection process based on scores, otherwise the page

Algorithm 3 Selector Mapper

```
1:  class SelectorMapper
2:      curTime ← context.getLong()                          ▷ Generator current time
3:      method Map( Url u , CrawlDatum d)
4:          if filter then                                   ▷ Check if there are URLFilters
5:              ApplyFilters(u, d, curTime)                  ▷ Apply the URLFilters to u
6:          if shouldFetch(u, d,curTime) then
7:          sort ← 1.0f
8:          sort ← generatorSortValue(u, d, sort)
9:          if Checks then          ▷ Checks if the page pass some tailored controls
10:         entry ← new SelectorEntry()                      ▷ Is a new writable object
11:         entry ← d
12:         entry ← u
13:         EMIT(sort, entry)
```

will be reject. Sort ,defined at row 7, is a float object where will put score information using *generatorSortValue*. After that we apply some checks, to information contained in *d* related to a certain page. For example we can compare the *Score* of the page to a certain threshold. Latest we define a new variable called entry that is an object of SelectorEntry, this java class is defined as *implements* and it implement a Writable interface that internally it have:

- **url** = A Text object that stores text using standard UTF8 encoding. It provides methods to serialize, deserialize, and compare texts at byte level. The type of length is integer and is serialized using zero-compressed format.
- **datum** = A CrawlDatum object, well explained at the beginning of the report.
- **segnum** = A IntWritable object, initialize at 0, that is a WritableComparable for ints.

At the end of the algorithm we produce a pair formed by sort and entry , as we can see in *Table 6.*

sort	entry
10.0	outputurl=https://www.amazon.it/ datum=Version: 7 Status: 1 (db_unfetched) Fetch time: Thu Mar 21 20:07:55 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000 seconds (30 days) Score: 10.0 Signature: null Metadata: _ngt_=1553358339965, segnum=0
1.0	outputurl=https://www.ansa.it/ datum=Version: 7 Status: 1 (db_unfetched) Fetch time: Thu Mar 21 20:07:55 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000 seconds (30 days) Score: 1.0 Signature: null Metadata: _ngt_=1553358339965 , segnum=0

Table 6. Some map output parameters

Algorithm 4 SelectorReducer

```
1:  class SelectorReducer
2:      method Reduce( Sort s , entry[e1, e2, ..])
3:          count ← 0
4:          currentsegmentnum ← 0
5:          segCounts =new [maxNumSegments]
6:          for values ∈entry[e1, e2, ..] do
7:              if count==topN then
8:                  if currentsegmentnum < maxNumSegments then
9:                      currentsegmentnum←currentsegmentnum +1
10:                     count ← 0
11:                 else
12:                     Break
13:             segCounts← segCounts+1
14:             values ← currentsegmentnum          ▷ Update segnum in metadata
15:             count← count+1
16:         EMIT(s , values )
```

The *SelectorReducer* deals with the generation of segment, that will be used to the *Fetchet* module, in the reduce phase we must consider the arguments declared in the "command" for the creation of the *segment*, showed before. The *Reduce* method takes in input a *FloatWritable* and *Iterable<SelectorEntry>*, they are the variables sort and entry outputted by the Mapper. At the beginning of the *Reduce* method we have to initialize some variables: count deals with of the number of URL, *currentsegmentnum* tell us in witch segment we store a certain URL and *segCounts* count the number of segments. After that we enter in a for cycle where we assign a certain segment to a *values* that is a SelectorEntry object, the following *if* are responsible to check and update the variables use to manage the segments. In row 14 we update the field *segnum* contained in *values* object, later we increment the count variable and we produce a pair formed by sort, *entry*. As we can see in *Table 7*, these are the outputted data obtained from the Reducer and since we have been set **topN** equal 2 in our segment we have just only the two URL with the high score.

sort	entry
10.0	outputurl=https://www.amazon.it/ datum=Version: 7 Status: 1 (db_unfetched) Fetch time: Thu Mar 21 20:07:55 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000 seconds (30 days) Score: 10.0 Signature: null Metadata: _ngt_=1553361046923, segnum=1
7.0	outputurl=https://www.google.it/ datum=Version: 7 Status: 1 (db_unfetched) Fetch time: Thu Mar 21 20:07:55 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000 seconds (30 days) Score: 7.0 Signature: null Metadata: _ngt_=1553361046923 , segnum=1

Table 7. Reduce output parameters

Another Map-Reduce is use for "modify" output data obtained just before, in other words we change the couple *key*, *value*. In *SelectorInverseMapper* we create a new couple formed by key and entry, in the first one we store the URL and the second remains unchanged from the input.

Algorithm 5 SelectorInverseMapper

```
1: class SelectorInverseMapper
2:     method Map(sort s, entry)
3:         key← entry.url()
4:         EMIT(key, entry)
```

In *PartitionReducer* takes in input object Text and SelectorEntry and he outputted a couple formed by Text and CrawlDatum. The output couple contain an URL and is associate information.

Algorithm 6 PartitionReducer

```
1: class PartitionReducer
2:     method Reduce(key, entry [e1, e2, ..])
3:         for values ∈entry[e1, e2, ..] do
4:             EMIT(values.url() ,values.date())
```

We can see in *Table 8* the outputted result of *PartitionReducer*. So in the end of Generator module we have produced just one segment that will be used by Fetcher module.

key	value
https://www.google.it/	Version: 7 Status: 1 (db_unfetched) Fetch time: Thu Mar 21 20:07:55 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000 seconds (30 days) Score: 7.0 Signature: null Metadata:_ngt_=1553365432638
https://www.amazon.it/	Version: 7 Status: 1 (db_unfetched) Fetch time: Thu Mar 21 20:07:55 CET 2019 Modified time: Thu Jan 01 01:00:00 CET 1970 Retries since fetch: 0 Retry interval: 2592000 seconds (30 days) Score: 10.0 Signature: null Metadata:_ngt_=1553365432638

Table 8. PartitionReducer output

4 FETCHER

4.1 Setup and testing

Fetcher is the third module of Nutch crawler, into the *segment* folder he create the *crawl_fetch* that contains contains status reports from the fetching, that is, whether it was successful, what was the response code, etc. This is stored in a map file of <url, CrawlDatum>. In Eclipse we have to **org.apache.nutch.fetcher** package and find the **Fetcher** java class, after that you have to configure the "run configuration".For run the fetcher we use: *bin/nutch fetch segments/20190323172544*
For better clear up will present all the possible arguments that we can set up:

- **segment**: This is the path to the previously generated segment directory we wish to fetch
- **threads n**: This argument invokes the number of threads we wish to work concurrently on fetching URLs in the desired segment e.g. the number of fetcher threads the fetcher should use. This is also determines the maximum number of requests that are made at once (each fetcher thread handles one connection).
- **D...**: overwrite a Nutch/Hadoop property from command-line, e.g.
- **Dfetcher.parse=true**: Make fetcher parse documents, overwriting the default value defined in nutch-default.xml or the setting in nutch-site.xml.

Using well configured VM argument will be useful for understand how does it work the module. The subparagraph MapReduce algorithm will be avoided because the Fetcher module don't use the MapReduce pattern.

5 PARSER

5.1 Setup and testing

Parser is the fourth module of Nutch crawler, into the *segment* folder he create the *crawl_parse* that contains the list of outlinks for each successfully fetched and parsed page is stored here so that Nutch can expand its crawling frontier by learning new URLs. In Eclipse we have to **org.apache.nutch.parse** package and find the **ParseSegment** java class, after that you have to configure the "run configuration".For run the fetcher we use: *bin/nutch parse segments/20190323172544*
For better clear up will present all the possible arguments that we can set up:

- **segment**: This should be the path to the segment directory containing our data for parsing.
- **noFilter**: Optional flag to NOT filtering URLs.
- **noNormalize**: Optional flag for NOT normalizing URLs.

Using well configured VM arguments and java LOG in Eclipse will be useful for understand how does it work the module.

5.2 MapReduce algorithm

The *Map* method takes in input URL and Content, this last contains the html code of the page. At the beginning of *Map* we define *ParseResult* that is an object of the class ParseResult. The class ParseResult is defined as *implements* and it implements an Iterable<Map.Entry<Text, Parse» and internally have many methods to manage and store the result of a parse. *ParseUtil* is an object o class ParseUtil and this class containing methods to simply perform parsing utilities such as iterating through a preferred list of URLs to obtainparse objects. At the first *if* we check if *ParseUtil* is null, if yes it will be initialize *ParseUtil* and after we assign to *ParseResult* the result of parsing for a certain page, more in detail it contains pairs formed by url, parse data. After that we enter in a for cycle where we assign to *u* variable the value contained in the entry key and we assign to *parse_end* the outlink, parse text that is a text coming from the parsing of a certain page plus some metadata and we can see and example of *parse_end* in *Figure 2*. In the end we obtain the couple *u*, *parse_end*.

Algorithm 7 ParseSegmentMapper

```
1:  class ParseSegmentMapper
2:      method Map(Url u, Content c)
3:          ParseResult ← null
4:          ParseUtil ← null
5:          if ParseUtil==null then
6:              ParseUtil ← get.initialize()          ▷ we update it with the configuration
7:              ParseResult ← ParseUtil.parse(content)          ▷ result of parsing a page
8:          for entry ∈ ParseResult[pr_1, pr_2, ..] do      ▷ entry is hashmap <url, parseResult>
9:              u ← entry.getKey()
10:             parse ← entry.getValue()
11:             parse_end ← parse.getinfo()          ▷ insert outlink, parse text, metadata
12:     EMIT(u, parse_end)
```

Figure 2 shows the information stored in *parse_end*, as we can see we have "extract" from https://www.Google.it 21 out links with their associated anchor text plus several content metadata and after that we have the parsed text.

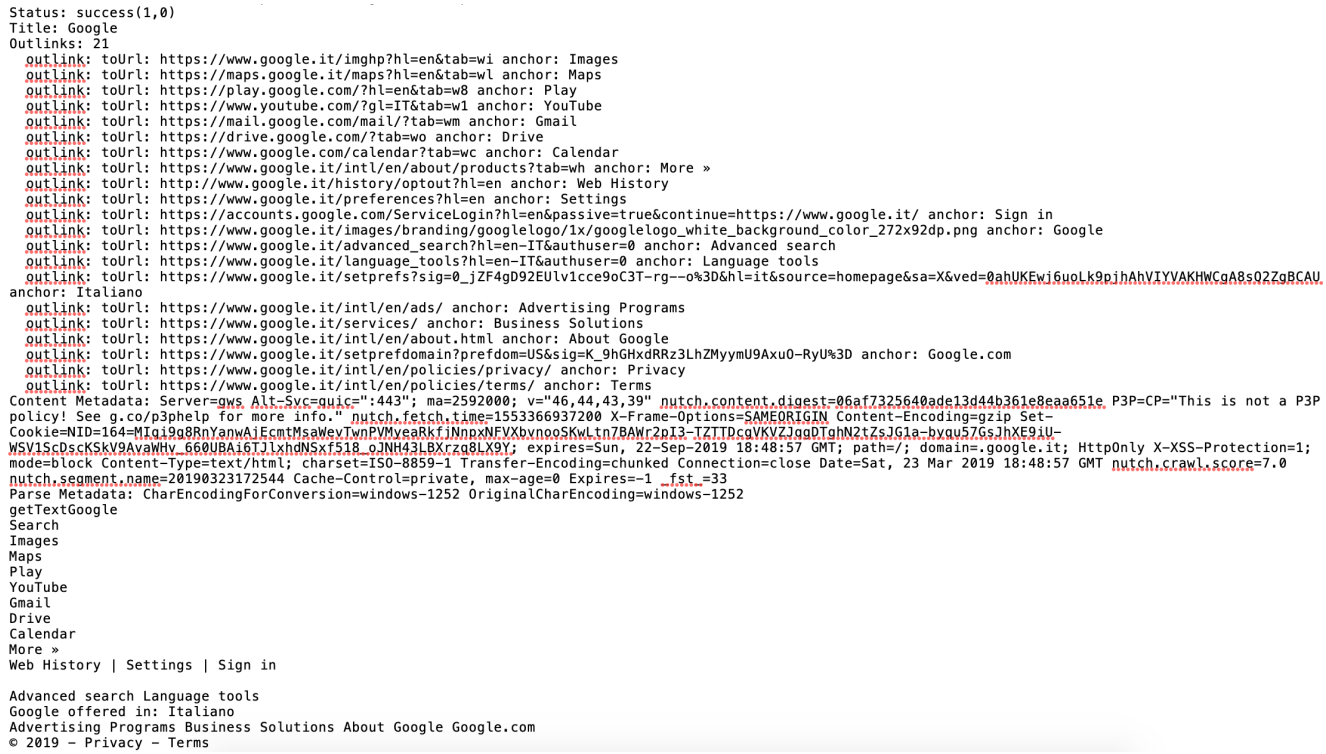

Fig. 2. Parse_end infromation

Algorithm 8 ParseSegmentReducer

```
1: class ParseSegmentReducer
2:     method Reduce(Url u, parse_end[p1, p2, ..] )
3:         for values ∈ parse_end[p1, p2, ..] do
4:             EMIT(u, values)
```

The *ParseSegmentReducer* collects the data emitted by the Mapper and the final results of *Parse* are three folders that contain:

- parse_data: Metadata collected during parsing; among others, the list of outgoing links (out- links) for a page. This information is crucial later on to build an inverted graph (of incoming links—inlinks).
- parse_text: Plain-text version of the page, suitable for indexing in Lucene. These are stored as a map file of <url, ParseText> so that Nutch can access them quickly when building summaries (snippets) to display the list of search results.
- crawl_parse: contains the list of outlinks for each successfully fetched and parsed page is stored here so that Nutch can expand its crawling frontier by learning new URLs

6 LINK INVERTER

6.1 Setup and testing

Link Inverter is the fifth module of Nutch crawler and he creates/initializes LinkDb, this database stores the incoming link information for every URL known to Nutch. It is a map file of<url, Inlinks>, where Inlinks is a list of URL and anchortext data. It's worth noting that this information is not immediately available during page collection, but the reverse information is available, namely that of outgoing links from a page.In Eclipse we have to **org.apache.nutch.crawl.LinkDb** package and find the **LinkDb** java class, after that you have to configure the "run configuration". For run the fetcher we use: *bin/nutch invertlinks LinkDb segments/20190323172544.*
For better clear up will present all the possible arguments that we can set up:

- **linkdb**: This should be the path the the output linkdb to create or update.
- **dir <segmentsDir>**: This corresponds to the parent directory containing several segments, OR
- **<seg1> <seg2> ...**: A list of segment directories to create a inverted linkdb from.
- **force**:This arguement forces an update even if linkdb appears to be locked
- **noNormalize**:We pass this if we don't normalize link URLs. This obtains us a true representation of incoming links within the linkdb.

- **noFilter**:This parameter avoids and doesn't apply any of our current URLFilters to link URLs.

Using well configured VM arguments and java LOG in Eclipse will be useful for understand how does it work the module

6.2 MapReduce algorithm

This "class" is in charges to create a LinkDB from a segment that was already parsed. The LinkDb is an inverted link map that listing incoming links for each url, for doing this we use the Mapper of Hadoop framework. *LinkDbMapper* takes in input an URL *u* and the associated parse data *p*. ParseData is the class from which comes the object *p* and it extends VersionedWritable(a base class for Writables that provides version checking). The main goal of ParseData is contain data extracted from a page's content. At the beginning we define the variable *fromUrl*, that tell us from which URL coming from the *p*, after that we check if there are some Normalizers/Filters to apply at the url stored in this variable, this filters and "normalization rules" and have defined in the configuration files.

Algorithm 9 LinkDbMapper

```
1:  class LinkDbMapper
2:      method Map(Url u, parseData p)
3:          fromUrl← key
4:          if urlNormalize then
5:              fromUrl←Normalize(fromUrl)
6:          if urlFilters then
7:              fromUrl←Filters(fromUrl)
8:          outlinks_array[]←p.getOutlinks()
9:          in ←new Inlinks()
10:         for i < outlinks_array.length(), i++ do
11:             outLink ← outlinks_array[i]
12:             toUrl ← outLink.getToUrl()
13:             if urlFilters then
14:                 toUrl ←Filters(toUrl)
15:             anchor ←outLink.getAnchor()
16:             if anchor.length() > maxAnchorLength then
17:                 anchor←anchor.substring()
18:             in ← in.add(fromUrl, anchor)
19:             EMIT(toUrl, in)
```

We store in outlinks_array (it is an array) all the out links that was contained in *p* and we create a new variable *in* where will be stored fromUrl and anchor. In the for cycle we iterate over the elements (out links) contained in outlinks_array and one by one they are assigned to the variable *toUrl*, later we check if there are some filters to apply and then is define a new variable, *anchor*, where we will be stored the

anchor text, which is a portion of text that is linked to another page, i.e hyperlink. After we check the anchor text length and if is greater then *maxAnchorLength* we make it smaller, in the end we store the fromUrl and anchor in *in* variables and we produce as output of the map a couple formed by *toUrl* and *in*.
This couple is show in *Table 9*, so we have created an index that focusing on the number of inbound links of a web page.

toUrl	inLinks
http://www.facebook.com/pages/ANSAit/158259371219	fromUrl: https://www.ansa.it/ anchor: Facebook
http://www.ansamed.info/	fromUrl: https://www.ansa.it/ anchor:Mediterraneo

Table 9. Map output

7 PAGERANK AND LINKRANK, OPIC

In this paragraph we compare OPIC, LinkRank with PageRank only form the theoretical point of view, the first two technique are supplied by Nutch. **PageRank** results from a mathematical algorithm based on the webgraph, created by all World Wide Web pages as nodes and hyperlinks as edges, taking into consideration authority hubs such as cnn.com or usa.gov. The rank value indicates an importance of a particular page. A hyperlink to a page counts as a vote of support. The PageRank of a page is defined recursively and depends on the number and PageRank metric of all pages that link to it ("incoming links"). A page that is linked to by many pages with high PageRank receives a high rank itself. In the below formula we can see how the PageRank of the page i is calculate.

$$r(i) = \alpha \sum_{j \in B(i)} \frac{r(j)}{\#out(j)} + (1-\alpha)\frac{1}{N} \quad \forall i \tag{1}$$

For sake of clarity we present in detail all the parameter in the formula:

- **B(i)**: Is the set of pages that link to i i.e incoming link.
- **#out(j)**: Is the number of link exiting from j i.e outlink.
- **(1-α)**: Teleportation step is the probability to go to another page.
- **(N)**: Number of nodes in the graph.

LinkRank can be found in **org.apache.nutch.scoring.webgraph.LinkRank.**. perform an iterative link analysis. LinkRank is a PageRank-like link analysis program that converges to stable global scores for each url. Similar to PageRank, the LinkRank program starts with a common score for all urls. It then creates a global score for each url based on the number of incoming links and the scores for those links

and the number of outgoing links from the page. The process is iterative and scores tend to converge after a given number of iterations. It is different from PageRank in that nepotistic links such as links internal to a website and reciprocal links between websites can be ignored. The number of iterations can also be configured; by default 10 iterations are performed. Unlike the previous OPIC scoring, the LinkRank program does not keep scores from one processing time to another. The web graph and the link scores are recreated at each processing run and so we don't have the problems of ever increasing scores. The formula used to compute LinkRank is equal to PageRank formulas but as we can said before LinkRank don't take into account links internal to a website and reciprocal links between websites so seems very easy modify the code of the LinkRank class for include in the computation the avoided internal and reciprocal links .

OPIC, On-Line Page Importance Computation, computes the importance of pages on-line, with limited resources, while crawling the web. It can be used to focus crawling to the most interesting pages. Moreover, it is fully integrated in the crawling process, which is important since acquiring web pages is the most costly part of the system. Intuitively speaking, some "cash" is initially distributed to each page and each page when it is crawled distributes its current cash equally to all pages it points to. This fact is recorded in the history of the page. The importance of a page is then obtained from the "credit history" of the page. The intuition is that the flow of cash through a page is proportional to its importance. It is essential to note that the importance we compute does not assume anything about the selection of pages to visit. If a page "waits" for a while before being visited, it accumulates cash and has more to distribute at the next visit. Nutch provide the class **org.apache.nutch.scoring.opic.OPICScoringFilter** to compute OPIC, but in the latest version of Nutch it is no longer available. In the below formula we can see how static OPIC is calculate.

$$C(k) = \frac{C(k) + H(k)}{G + 1} \quad \forall k \tag{2}$$

For sake of clarity we present in detail all the parameter in the formula:

- **C(k)**: Cash of page.
- **G**: is the total score of the distributed cash for the whole graph.
- **H(k)**: History of page.

We can se and example of OPIC at this link https://wiki.apache.org/nutch/FixingOpicScoring?action=AttachFile&do=get&target=opic.pdf and as we can see from example OPIC is just an alternative to the power method to compute the largest eigenvector and so it can be applied to compute the HITS, Hyperlink-Induced Topic Search.

```
OPIC:
On-line Page Importance Computation
for each i let C[i] := 1/n ;
for each i let H[i] := 0 ;
let G:=0 ;
do forever
begin
  choose some node i ;
  %% each node is selected
  %% infinitely often
  H[i] += C[i];
  %% single disk access per page
  for each child j of i,
    do C[j] += C[i]/out[i];
  %% Distribution of cash
  %% depends on L
  G += C[i];
  C[i] := 0 ;
end
```

Fig. 3. OPIC algorithm[2]

In *Figure 3* we have the algorithm of OPIC, as we can see changing/modifying the appropriate part of the code we can easily obtain the PageRank formula.

8 CONCLUSION

As we can see in almost all modules of Nutch we use Hadoop MapReduce for manipulate large amount of data and implement various complex processing tasks typical for a crawler. Once understand how the "commands set" works Nutch is easy to use and in addition he is suitable for huge customization very useful to facilitate the desire of the developers and keeping always update the crawler functionality. The main advantages achieve by Nutch are :

- Highly scalable and relatively feature rich crawler.
- Quality, crawling can be biased to fetch "important" pages first.
- Highly Customizable, because we can integrate into the crawler our home made code.
- Suitable for calculate PageRank, HITS.

REFERENCES

[1] *Figure 1*, `https://i.stack.imgur.com/3RVif.png`
[2] *Figure 3*, `http://www2003.org/cdrom/papers/refereed/p007/p7-abiteboul.html`