SoftwareX 6 (2017) 98–106

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

RCrawler: An R package for parallel web crawling and scraping

Salim Khalil *, Mohamed Fakir

Department of Informatics, Faculty of Sciences and Technics Beni Mellal, Morocco

ARTICLE INFO

Article history:
Received 8 November 2016
Received in revised form 2 March 2017
Accepted 13 April 2017

Keywords:
Web crawler
Web scraper
R package
Parallel crawling
Web mining
Data collection

ABSTRACT

RCrawler is a contributed R package for domain-based web crawling and content scraping. As the first implementation of a parallel web crawler in the R environment, RCrawler can crawl, parse, store pages, extract contents, and produce data that can be directly employed for web content mining applications. However, it is also flexible, and could be adapted to other applications. The main features of RCrawler are multi-threaded crawling, content extraction, and duplicate content detection. In addition, it includes functionalities such as URL and content-type filtering, depth level controlling, and a robot.txt parser. Our crawler has a highly optimized system, and can download a large number of pages per second while being robust against certain crashes and spider traps. In this paper, we describe the design and functionality of RCrawler, and report on our experience of implementing it in an R environment, including different optimizations that handle the limitations of R. Finally, we discuss our experimental results.

Code metadata

Current code version	v 0.1
Permanent link to code/repository used for this code version	http://github.com/ElsevierSoftwareX/SOFTX-D-16-00090
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	r, Java
Compilation requirements, operating environments & dependencies	64-bit operating system & R environment version 3.2.3 and up (64-bit) & R packages: httr, rJava, xml2, data.table, foreach, doParallel, parallel
If available Link to developer documentation/manual	https://github.com/salimk/Rcrawler/blob/master/man/RcrawlerMan.pdf
Support email for questions	khalilsalim1@gmail.com

1. Introduction

The explosive growth of data available on the World Wide Web has made this the largest publicly accessible data-bank in the world. Therefore, it presents an unprecedented opportunity for data mining and knowledge discovery. Web mining, which is a field of science that aims to discover useful knowledge from information that is available over the internet, can be classified into three categories, depending on the mining goals and the information garnered: *web structure mining*, *web usage mining*, and *web content mining* [1]. *Web structure mining* extracts patterns from the linking structures between pages, and presents the web as a directed graph in which the nodes represent pages and the directed edges represent links [2]. *Web usage mining* mines user activity patterns gathered from the analysis of web log records, in order to understand user behavior during website visits [3]. *Web content mining* extracts and mines valuable information from web content. The latter is performed with two objectives: *search results mining*, which mines to improve search engines and information retrieval fields [4]; and *web page content mining*, which mines web page content for analysis and exploration purposes. This work is part of a project that aims to extract useful knowledge from online newspaper contents. In web content mining, data collection constitutes a substantial task (see Fig. 1). Indeed, several web mining applications use web crawlers for the process of retrieving and collecting data from the web [5].

Web crawlers, or spiders, are programs that automatically browse and download web pages by following hyperlinks in a

* Corresponding author.
E-mail addresses: khalilsalim1@gmail.com (S. Khalil), fakfad@yahoo.fr (M. Fakir).

http://dx.doi.org/10.1016/j.softx.2017.04.004

Fig. 1. Schematic overview of the processes involved in web content mining applications.

methodical and automated manner. Various types of web crawlers exist. Universal crawlers are intended to crawl and index all web pages, regardless of their content. Others, called preferential crawlers, are more targeted towards a specific focus or topic [6]. Web crawlers are known primarily for supporting the actions of search engines, particularly in web indexing [7]. However, web crawlers are also used in other applications that are intended to collect and mine online data, such as web page content mining applications.

In order to enhance the accuracy of content mining results, only valuable data should be extracted. Certain irrelevant data, such as navigational bar banners and advertisements, should be excluded, and this involves a data extraction process. Data extraction, or data scraping [8], is the problem of extracting target information from web pages to produce structured data that is ready for post-processing.

Web crawling and data extraction can be implemented either as two separate consecutive tasks (the crawler fetches all of the web pages into a local repository, then the extraction process is applied to the whole collection), or as simultaneous tasks (while the crawler is fetching pages the extraction process is applied to each page individually). A web crawler is usually known for collecting web pages, but when a crawler can also perform data extraction during crawling it can be referred to as a web scraper. This paper describes the architecture and implementation of RCrawler, an R-based, domain-specific, and multi-threaded web crawler and web scraper.

This study is motivated by the need to build an enhanced R-base web crawler that can crawl and scrape web page content (articles, titles, and metadata) in an automated manner to produce a structured dataset. The difficult aspect of this was the implementation of a parallel crawler in an environment that is mainly dedicated to calculations and statistical computing, rather than automatic data processing. Thus, to enable data collection within the R environment, our challenge was to overcome these weaknesses and adapt the environment to our requirements.

R is a highly effective software environment for statistical analysis and data processing, and provides powerful support for web mining [9]. In fact, R provides a large set of functions and packages that can handle web mining tasks [10]. There are R packages available for data collection processes, such as *Rvest*, *tm.plugin.webmining*, and *scrapeR*. However, these packages do not provide basic crawling, because they can only parse [11] and extract contents from URLs, which the user must collect and provide manually. Therefore, they are not able to traverse web pages, collecting links and data automatically. For instance, in most cases users rely on external tools for performing this task. Hence, we incorporate the crawling process into the R environment, in order to offer a full data flow platform including the steps both before and after the actual data analysis. In fact, from a given URL *RCrawler* can automatically crawl and parse all URLs in that domain, and extract specific content from these URLs that matches the user criteria. Table 1 provides a comparison of some popular data collection packages, and illustrates the utility of our new package.

As described in Table 1, *scrapeR* and *rvest* require a list of URLs to be provided in advance. Meanwhile, *tm.plugin.webmining* can obtain some improvement, because it can fetch URLs from certain feed formats such as XML and JSON, but its usage is still limited by the fact that not all websites have feeds, and even if a feed exists it may not contain the full website tree. All of these packages can retrieve and parse specific web pages efficiently. However, the common weakness of these tools is that they are limited in handling multiple requests, and any attempt to loop over the same function for a list of URLs may result in errors or crashes on account of the politeness constraints.

RCrawler can easily be applied to many web content mining problems, such as opinion mining [12], event or topic detection [13], and recommender systems [14]. Moreover, its functionality allows it to be expanded to explore specific website structures. However, its usage is still limited to small projects, owing to the nature of the R environment, which is not fully dedicated to handling massive data crawling.

This paper attempts to address many questions that beginning researchers often have regarding web content mining and data collection techniques, regardless of the programming language used. In fact, we have tried to collect and summarize some techniques involved in the data collection process. In addition, we report all of the main challenges that any researcher or programmer may encounter when designing their first web crawler/scraper.

The remainder of this paper is structured as follows. Section 2 presents the key technical requirements that guided the crawler development. Section 3 presents the crawler design and architecture. Section 4 covers the main RCrawler features and their implementation details. Section 5 presents the main components of the package, and a practical demonstration of it. Section 6 describes usage patterns for RCrawler, and in Section 7 our experimental results are described, as well as a discussion of the performance measures and statistics of RCrawler. Finally, Section 8 presents our conclusions and directions for future research.

2. Goals and requirements

In this chapter, we describe the functional requirements and system goals that have guided our crawler implementation. There are five primary requirements:

1. *R-native*: Usually, when R-users need to crawl and scrape a large amount of data automatically, they turn to external tools to collect URLs or carry out the complete task, and then import the collected data into R. Thus, our main reasons for writing RCrawler were to support web crawling and scraping in the R environment. Thus, our solution should be natively implemented in R.
2. *Parallelization*: RCrawler should take advantage of parallelism, in order to gain significant performance enhancements and make efficient use of various system resources, including processors.
3. *Politeness*: The crawler should be able to parse and obey *robots.txt* commands. In addition, the crawler should avoid requesting too many pages in a short interval of time from a given host.
4. *Efficiency*: Our solution should make clever use of resources and be resilient to spider traps. The crawler should be able to detect and avoid duplicate URLs or web pages.
5. *Flexibility and Reconfigurability*: We would like to design a flexible system that can be applied in various scenarios. Below is a summarized list of the settings options for RCrawler:

    - Project name and directory.

Table 1
Comparison of some popular R packages for data collection.

Package Name	Crawl	Retrieve	Parse	Description
scrapeR	No	Yes	Yes	From a given vector of URLs, retrieves web pages and parses them to pull out information of interest using an XPath pattern.
tm.plugin.webmining	No	Yes	Yes	Follows links on web feed formats like XML and JSON, and extracts contents using boilerpipe method.
Rvest	No	Yes	Yes	Wraps around the *xml2* and *httr* packages so that they can easily to download and then manipulate HTML and XML.
RCrawler	Yes	Yes	Yes	Crawls web sites and extracts their content using various techniques.
Some basic web toolkits:				
XML, XML2	No	No	Yes	HTML / XML parsers
jsonlite, RJSONIO	No	No	Yes	JSON parser
RSelenium	No	No	Yes	Browser Automation
Selectr	No	No	Yes	Parses CSS3 Selectors and translates them to XPath 1.0
Httr, RCurl	No	Yes	No	Handles HTTP / HTTPS requests

- User agent, connection time-out, and request delay.
- Filters: Include/exclude content type (MIME), error pages, file extension, and URLs matching a specific regular expression pattern.
- Parallel crawling: the user specifies the number of nodes and number of connections (simultaneous requests).
- Choose to honor *Robots.txt* file or to ignore it.
- Maximum depth level to control the crawling depth.
- The crawler should provide the ability to apply some specific functions of content extraction, such as XPath patterns, to each crawled page during the crawling process.

Regarding visualization and results, the crawler should return the crawled data in data structures that are well-organized and ready to use, such as vectors, lists, and data frames. In fact, data returned by other packages is always outspread, and additional effort is required to arrange this into one data structure. We describe the data structures for RCrawler as follows:

- A data frame representing the generic URL index, including the list of fetched URLs and page details (content type, HTTP state, number of out-links and in-links, encoding type, and level).
- A file repository that contains all downloaded pages.
- A vector for scraped contents.
- A message including the crawling statistics.
- For link analysis, a data structure are needed to represent the connectivity of the web graph (edges).
- During the crawling process, the crawler should display the crawling state.

3. RCrawler architecture

Web crawlers must deal with several challenges simultaneously, some of which contradict with each other [15]. Inspired by previous work, such as Mercator [16] and Ubicrawler [17], we have attempted to make the crawler architecture as optimized and simple as possible in order to avoid overloading the host environment (see Fig. 2).

The crawler begins from a given website URL, provided by the user, and progressively fetches this and extracts new URLs (out-links). These in turn are added to the frontier list to be processed. The crawling process stops when all URLs in the frontier are processed.

First, the crawler initiates the working environment, comprising the index structure, the repository that will contain the collection of web documents, and the cluster nodes (workers) for parallel computing. Crawling is performed by multiple worker threads, and the *work-pool-handler* component prepares a pool of URLs to be processed in parallel. Then, each node executes the following functions for the given URL:

1. Download the corresponding document and its HTTP header using a GET request.
2. Parse and extract all links contained in the document.
3. Proceed with the canonicalization and normalization of URLs.
4. Apply a URL filter, keeping only URLs that match the user-supplied configuration, file type, and domain specific URLs.

As a result, for each URL each worker returns its HTML document, HTTP header details, and the list of discovered out-links. The *URL-seen* function checks if the URL has already been processed or queued before being added it to the frontier. Before storing the document in the repository, the *Is-not-duplicate?* function checks that it has not been processed with a different URL. Otherwise, it is discarded. We prefer to write the frontier list and the index to disk in each crawler iteration, to protect them against data loss in the case of a crawler crash. The following are the main RCrawler components:

- *HTTP requests Handler*: handles HTTP requests.
- *Link Extractor*: parses and extracts links from crawled documents.
- *Content duplicate checker*: detects duplicate and near-duplicate documents.
- *Work-thread manager*: handles multi-threading and parallel computing.
- *Data extraction*: a component for parsing and extracting web page contents.
- *Index*: a data structure to store information regarding crawled web pages.

Other components relating to spider traps, URL canonicalization, the *robot.txt* parser, and other features are discussed in later sections.

4. Functional properties and implementation

This section discusses the functionalities of the current release of RCrawler.

4.1. Parallel crawling — multithreading

The most straightforward way of improving the crawler performance and efficiency is through parallel computing [18]. However, the implementation of a multi-threaded crawler under R involves many limits and challenges.

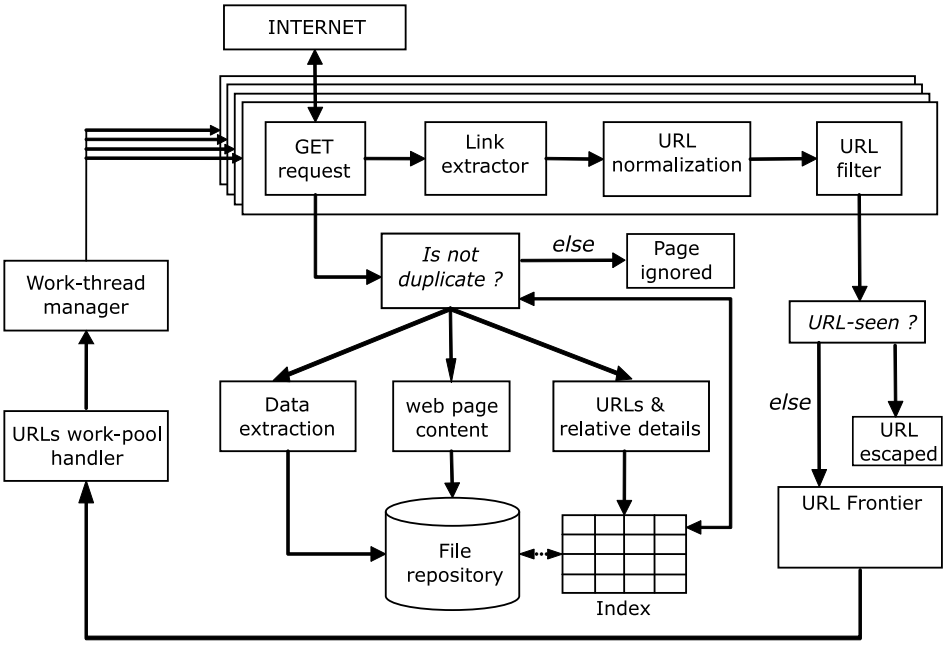

Fig. 2. Architecture and main components of R crawler.

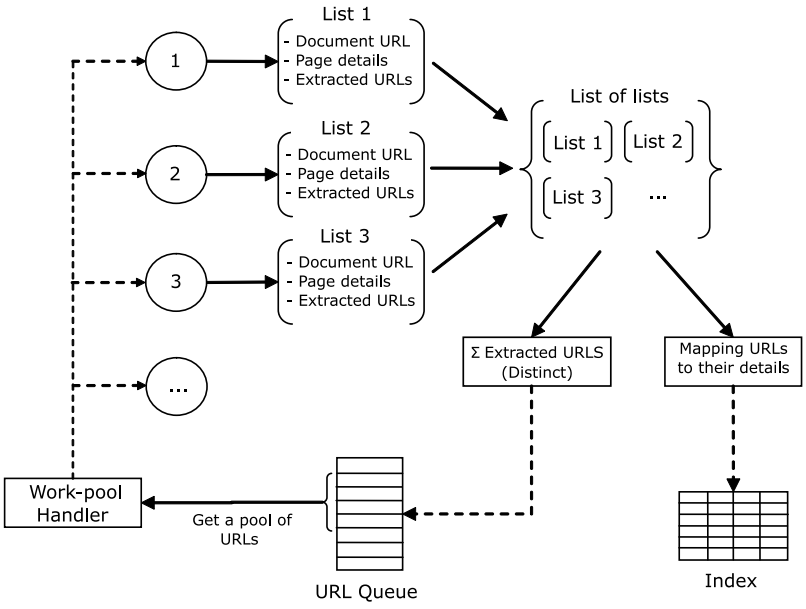

Fig. 3. Design of our multi-threading implementation in the R environment.

The process of a crawling operation is performed by several concurrent processes or nodes in parallel, as illustrated in Fig. 3. Essentially, each running node is assigned to a physical control thread provided by the operating system, and works independently. To handle parallel computation inside an R environment [19], we have used the *doParallel* and *parallel* packages [20,21]. First, we start up N node processes (with N specified by the user). Then, we initiate all nodes with shared libraries, functions, and data structures. However, because it is difficult to manage node access to shared data structures in R, we have limited this access to read-only. Therefore, all write operations are performed by the main crawling function. Thereafter, the *work-pool-handler* selects a URL pool from the frontier, to be processed in parallel by the nodes (the pool size is similar to the number of simultaneous requests). Note that the occurrence of too many simultaneous requests from one IP address may look like a DOS attack (frequent requests that prevent the server from serving requests from bona fide clients). To avoid such situations, it is recommended to limit the number of active requests to a given host. The *foreach* package [22] provides the basic loop structure, and utilizes the parallel back-end to execute the tasks and return a data structure representing the collection of results for all nodes. Note that the *foreach* loop returns a list of items in the same order as the counter, even when running in parallel. Thus, each node processes a given URL and returns a *list* data structure with three elements (the URL document, HTTP header details, and list of extracted URLs). These lists are subsequently returned together as a single collection (a collection of lists), to be isolated and mapped to their URL of origin by the main function (Fig. 2). The crawler should avoid requesting too many pages from a given host within a short time interval. Therefore,

we have introduced a *RequestDelay* function between subsequent requests to the same host. Finally, node processes are stopped and removed when crawling is complete.

4.2. HTTP requests handler

The HTTP requests handler is the component in charge of HTTP connections, and provides convenient functions for composing general HTTP requests, fetching URLs, getting and posting forms, and so on. An efficient crawler should provide a large degree of control over HTTP connections and the configurations of requests. In fact, it should allow the user to set some HTTP properties, such as the following:

User-agent: When crawling a website, RCrawler identifies itself by default as RCrawler. However, some web hosting companies may block a user agent if it is not a browser or if it is making too many requests. Thus, it is important to change the referrer (user agent) to simulate different browsers and continue crawling without being banned.

Timeout: The maximum request time, i.e., the number of seconds to wait for a response until giving up, in order to prevent wasting time waiting for responses from slow servers or huge pages.

Some additional header properties can be added to HTTP requests to control the manner in which the server handles incoming requests or to provide extra information regarding the requested content.

To improve the crawler's performance, we have limited the number of requests to one per URL. Therefore, the response must include the content and all of the relevant required information (response code, content type, encoding, etc.). The *httr* [23] package has been imported to manage HTTP connections, as this provides a rich set of functions for working with HTTP and URLs. In addition, it offers a higher-level interface using R socket connections.

Exception handling and error-checking are important considerations during the page fetching process, in order to prevent crashes at each failed request. In fact, the crawler checks response properties before parsing content. If it is null, then the domain name does not exist or could not be resolved. Otherwise, if the URL host server has issued an HTTP error code or a non-supported content type, then URL documents are not parsed, and according to the user configuration, this may or may not be indexed.

4.3. HTML parsing and link extraction

After fetching a web page, every crawling node needs to parse its content to extract hyperlinks and relevant information required for indexing. However, implementing parsing for content extraction inside nodes may slow down the crawling process and could result in an unbalanced load distribution.

Parsing is implemented using the *xml2* library [24], which is a simple and consistent interface built on top of the *libxml2* C library. Note that we only parse for hyper-links or specific data extraction through the XPath pattern. First, the content is retrieved from the HTTP request, and the entire document is parsed into a Document Object Model (DOM) [25] in a tree-like data structure in the C language. In this data structure, every element that occurs in the HTML is now represented as its own entity or as an individual node. All of the nodes together are referred to as the node set. The parsing process also includes an automatic validation step for malformation. In fact, *xml2* is capable of working on non-well-formed HTML documents, because it recognizes errors and corrects them to create a valid DOM. In the next step, the C-level node structure is converted into an object of the R language, through so-called handler functions, which manage transformations between C and R. This is necessary for further processing of the DOM. Finally, link extraction is performed by extracting all nodes that match *href* tags in the document, through the *xml_find_all* function. Thereafter, for each node we grab the *href* values using the *gsub* function.

4.4. Link normalization and filtering

The process of URL normalization [26] transforms the URL into its canonical form in order to avoid duplicate URLs in the frontier. In fact, normalization eliminates URLs that are syntactically different but point to the same page by, transforming them into syntactically identical URLs. A canonical URL comprises the following five rules (with examples):

1. Lowercase URL string
   from Http://www.Test.Com
   to http://www.test.com.
2. Resolve URL path
   from http://www.test.com/.././f.htm
   to http://www.test.com/f.htm.
3. Non-www URL to www
   from http://test.com/sub/sub/
   to http://www.test.com/sub/sub/.
4. Relative URLs to absolute base URL
   from /sub/file.html
   to http://www.test.com/sub/file.html.
5. Anchor elimination
   from http://www.test.com/f.html#tite
   to http://www.test.com/f.htm.

Almost all of these functions are implemented using regular expressions through pattern matching and replacement.

Following link normalization, extracted links must be filtered. At this particular stage, two levels of filtering are distinguished. The first affects the process of crawling, and is applied to links before being added to the frontier, to skip certain file types or paths. A prior form of identification of a file type is based on file extensions, and subsequent identification is confirmed through content-type response headers, which is more reliable. In addition, because RCrawler is a domain-specific crawler, extracted links should belong to the domain source, and all external links have to be removed. Another filtration method involves spider traps. In fact, some server-side scripts, such as CGI (common gateway interface), can generate an infinite number of logical paths (pages and directories), resulting in an infinitely deep website. Such spider traps can be avoided simply by checking the URL length (or the number of slashes in the URL). Finally, before adding a new URL to the work pool, we must check if the URL already exists in the frontier or if it has been fetched. The second level of filtering is based on the user settings and control links to be indexed by the crawler.

4.5. Duplicate and near-duplicate detection

There are many instances in which a web page may be available under multiple URLs. The probability of duplicate pages occurring on specific websites is higher for dynamic websites, such as blogs, forums, or e-commerce, owing to the URL's optional parameters. This will cause the web crawler to store the same document multiple times, which would negatively impact the content mining results. Therefore, to reduce redundant storage, processing costs, and network bandwidth requirements, a web crawler can perform a duplicate-detection test to determine whether or not a document has previously been stored [27]. A fast and efficient method of accomplishing this is to map each page's URL to a compact representative number, by using some hashing function that has a low probability of collisions, such as MD5 or SHA-1;

or by using Rabin's fingerprinting algorithm [28], which is much faster and offers strong probabilistic guarantees. We maintain a data structure called *fingerprints*, which stores the *checksums* of the downloaded document so that only fingerprints are compared, instead of comparing the entire document. However, there are certain situations where multiple URLs may present the same content visually, but differ structurally overall or in a small portion, such as links, script tags, or advertisements. The problem is that from a computer's point of view, pages are only the same if they exactly match each other byte-by-byte. These pages are considered near-duplicates. Charikar's fingerprinting [29], also known as *SimHash*, is a technique that helps to detect near-duplicates. This constitutes a dimensionality reduction method, which maps high-dimensional vectors to small-sized fingerprints. For web pages, this is applied as follows:

1. Define a fingerprint size, e.g., 64 bits.
2. The web-page is converted into a set of tokens (features). Each token is assigned a weight. In our implementation, we only tokenize the document, and we consider all tokens to have the same weight of 1.
3. Each token is represented by its hash value using a traditional hashing function.
4. A vector V (of length 64) of integers is initialized to 0. Then, for each token's hash values (h), the vector V is updated: the ith element of V is decreased by the corresponding token's weight if the ith bit of the hash value is 0. Otherwise, the element is increased by the corresponding token's weight.

To measure the similarity of two fingerprints A and B, we count the number of bits that differ between the two as a measure of dissimilarity. If the result is equal to 0, then the two documents are considered identical. We decided to implement three possible options in RCrawler:

1. Escape the duplicate detection process.
2. Duplicate detection using MD5 hashing.
3. Near-duplicate detection using *SimHash*.

For MD5 hashing, we used the *digest* package. However, we were not able to implement the *SimHash* algorithm in R, because of the limited integer size ($2 * 10^9$), which blocks calculation. Therefore, we decided to implement this in Java and then wrap it inside our RCrawler package to make it available in R. To interface R with Java, we used the *rJava* library [30], which provides all the necessary functions for invoking a Java class from inside an R function. As a result, we created two functions:

- *getSimhash* takes the web page as a string and hash bit size as input, and returns its fingerprint.
- *getDistance* takes two fingerprints as input and returns the similarity value.

4.6. Manage data structures

In memory, RCrawler is a domain-specific crawler that only traverses web pages of a particular domain name. Therefore, the frontier size is limited and, can fit into memory. Furthermore, R can allocate up to $2^34 - 1$ bytes (8 GB) of memory, and can support vectors with over $2^31 - 1$ elements (2 billion) on 64-bit-platforms, which is sufficient in our case. As a consequence, the frontier data structure is implemented as a character vector in memory. As crawling proceeds, URLs to be crawled next are selected sequentially by the *work-pool-handler* function, and newly discovered URLs are sent to the tail of the frontier. After fetching a web page, it is indexed in the crawler index, which is represented in memory as a data frame.

In repository, For each website, RCrawler initiates a local folder that will hold all the files for that website, which are as follows:

- *extractedcontent.csv*: contains extracted contents.
- *index.csv*: the crawler's index file.
- *pages*: collection of downloaded web pages.
- *backup*: the crawler state for restoring/resuming a crawling session.

4.7. Additional features implementation

Robot.txt ***parser***. Not everything that can be crawled should be crawled, and there are more and less polite ways of doing so. In fact, web-masters sometimes desire to keep at least some of their content prohibited from crawling, or in some cases they may want to block some crawlers to save bandwidth. The *robots.txt* file is used for this purpose. This robot exclusion protocol? tells the crawler which information on the site may be harvested, and which may not. For this purpose, we implemented a function that fetches and parses *robots.txt* by means of regular expressions, and identifies corresponding rules of access. When starting RCrawler, if the *Obeyrobots* parameter is set to TRUE, then the crawler will parse the website's *robots.txt* file and return its rules (allowed and disallowed directories or files) to be filtered.

Out-links/In-links counting. The number of a web page's out-links represents the number of links extracted and filtered from that page. However, the number of in-links (links from other pages that point to that page) is calculated during the crawling process. During the crawling process, the program looks for every extracted URL in the index, and increments its corresponding in-link value.

Depth level control. This is not the file depth in a directory structure, but rather it represents the distance between the root document and all extracted links. For example, level 0 represents the root document. Thus, after fetching all links in this document, the level is incremented by 1, and so on. We have implemented this feature to control the depth level of crawling.

Backup and resume a session. As a regular function on R, the crawler returns its results at the end of the crawling process (when all URLs in the frontier have been processed). Thus, to avoid data loss in the case that a function crashes in the middle of action, we need to save the crawler state and data structures on disk. However, writing all of the data to disk on each iteration is time-consuming, on account of disk-seeking and latency. Therefore, a convenient solution is to initiate a file connection and only append new entries to the existing file, instead of overwriting the entire file on each iteration. Another solution is to use global R variables. In fact, within the R computation mechanism environments play a crucial role, as R consistently uses them just behind the scenes for an interactive computation. The task of an environment is to associate, or bind, a set of names to a set of values. By default, when a function is called a local environment is created. Thus, all of its variables are associated with that environment, and can only be used for that function. In the case that it crashes, all of its dependent variables are wasted. However, by using the global environment (the top level environment that is available) to handle some critical data structures, such as the crawler index, the frontier will make these more reliable against function crashes. However, this is still far from the safety offered by persistent storage.

Content extraction. Web crawling and data extraction can either be implemented as two separate consecutive tasks (the crawler fetches all web pages into a local repository and then the extraction process is applied to the whole collection), or as simultaneous tasks (while fetching URLs, content scraping

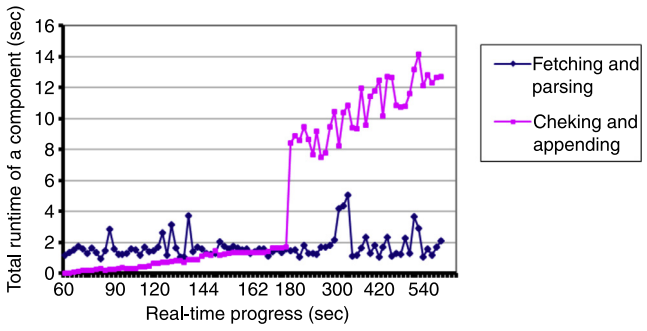

Fig. 4. As crawling proceeds, the matching and appending components require more time than other components.

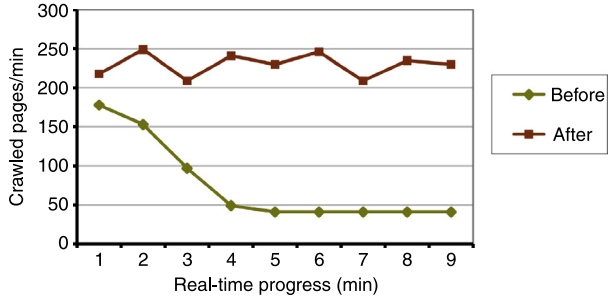

Fig. 5. After optimizing the matching and appending component, the crawler speed becomes stable.

is applied to each page individually). To extend the abilities of RCrawler as a web scraper, we introduced manual extraction using XPath patterns. Users can specify an unlimited number of named patterns to extract during the crawling process. Moreover, users can specify which pages should be extracted (detail pages), by filtering URLs matching a specific regular expression pattern using the *urlregexfilter* parameter.

4.8. Evaluation and improvement

During preliminary tests, we have noticed that the crawler speed decreases significantly after crawling 1000 pages. To identify the component that is slowing down the crawler, we have run tests on each component individually fetching, parsing, checking, and appending links to the data structure.

Fig. 4 illustrates the time required for each RCrawler component. It can be seen that the matching and appending processes take more time as the crawling proceeds. In fact, checking if the URL exists in the table consumes more time as the frontier size increases. To check if a URL exists in the frontier, we use the `%in%` operator. Unfortunately, this operator is not designed to handle large datasets. Therefore, in order to maintain the crawler speed, we have substituted the `%in%` operator with the `%chin%` operator from the *data.table* package [31], which is optimized for large datasets and character vectors. Furthermore, we have optimized the filtering and appending operations.

Fig. 5 shows the crawler speed before and after optimization. It can be seen that the number of downloaded web pages becomes more stable and linear. Note that the small remaining speed variations are due to servers and internet connections that can become overloaded and cause delays in the response time.

5. Package components

This version of the RCrawler package includes the following functions:

RCrawler: The crawler's main function.

```
Rcrawler(Website, no_cores, nbcon, MaxDepth, DIR,
    RequestsDelay = 0, duplicatedetect = FALSE, Obeyrobots
    = FALSE, IndexErrPages, Useragent, Timeout = 5,
    URLlenlimit = 255, urlExtfilter, urlregexfilter,
    ignoreUrlParams, statslinks = FALSE, Encod, patterns,
    excludepattern)
```

LinkExtractor: Takes a URL as input, fetches its web page, and extracts all links following a set of rules.

```
LinkExtractor(url, id, lev, IndexErrPages, Useragent,
    Timeout = 5, URLlenlimit = 255, urlExtfilter,
    statslinks = FALSE, encod, urlbotfiler, removeparams)
```

LinkNormalization: transforms a list of URLs into a canonical form

```
LinkNormalization(links, current)
```

Getencoding: Parses a web page and retrieves the character encoding based on the content and HTTP header.

```
Getencoding(url)
```

RobotParser: Fetches and parses *robots.txt* file and returns its corresponding access rules.

```
RobotParser(website, useragent)
```

GetSimHash: Generates SimHash fingerprint of a given web page, using an external Java class.

```
getsimHash(string, hashbits)
```

Linkparameters: Extracts URL parameters and values from a given URL.

```
Linkparameters(URL)
```

Linkparamsfilter: Excludes a given set of parameters from a specific URL.

```
Linkparamsfilter(URL, params)
```

Contentscraper: Extracts contents matching a given set of XPath patterns.

```
contentscraper(webpage, patterns, patnames, excludepat,
    astext = TRUE, encod)
```

6. Practical usage

In this section, we present some examples of how to use the package.

- `Rcrawler("http://www.example.com/")`

  Crawl, index, and store web pages using the default configuration.

- `Rcrawler(Website = "http://www.example.com/", no_cores =8, nbcon=8, Obeyrobots = TRUE, Useragent="Mozilla 3.11")`

  Crawl and index the website using 8 cores and 8 parallel requests with respect to *robot.txt* rules.

- `Rcrawler(Website = "http://www.example.com/", no_cores = 4, nbcon = 4, urlregexfilter = "/\\d{4}/\\d{2}/", DIR = "./myrepo", MaxDepth=3)`

  Crawl the website using 4 cores and 4 parallel requests. However, this will only index URLs matching the regular expression pattern (`/\\d{4}/\\d{2}/`), and stores pages in a custom directory "myrepo". The crawler stops when it reaches the third level.

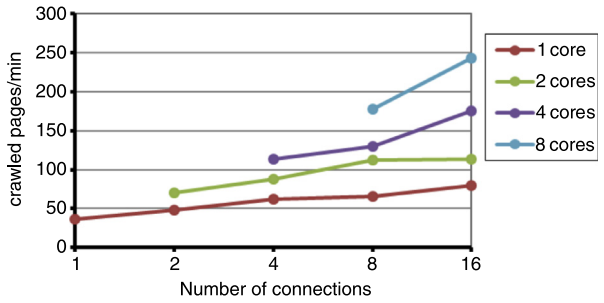

Fig. 6. The number of crawled pages/min increases when using more processes and connections.

- `Rcrawler(Website = "http://www.example.com/", urlregexfilter = "/\\d{4}/\\d{2}/", patterns = c("//*[@class='post-body entry-content']","//*[@class='post-title entry-title']"))`

  Crawl the website using the default configuration and scrape content matching two XPath patterns from web pages matching a specific regular expression. Note that the user can use the *excludepattern* parameter to exclude a node from being extracted, e.g., in the case that a desired node includes (is a parent of) an undesired "child" node.

7. Experimental results and discussion

We run our experiments on a common PC, with 6 GB RAM and a 100 GB SSD hard drive, where the CPU is a core i7 and the operating system is Microsoft Windows 8.1. Our package is installed on an R platform version 3.2.2. Pages are collected from the *New York Times* website.

7.1. Evaluating RCrawler multi-threading

After loading the RCrawler package, we test run the crawler under various configurations, varying the number of cores between one and eight and the number of connections (simultaneous requests) between one and 16. Note that we have disabled extra features, such as extraction and filtering. Fig. 6 presents the average number of crawled pages per minute for each scenario. By observing the results, it is evident that the crawler speed increases for more cores and connections.

7.2. Evaluating RCrawler speed performance

To make a fair comparison, we have selected three similar or near-identical open source projects for different platforms. Table 2 presents a comparison of open source crawlers regarding the language used, essential role, and parallelization support. *Scrapy* is an open source Python application framework for writing web spiders that crawl websites. This is commonly regarded as the fastest open source web scraper. However, there is no convincing research to prove this. In contrast to other crawlers, *Scrapy* also allows the extraction of data from web pages during the crawling process. This is a crucial feature for providing meaningful results in our evaluation. To compare the speeds achieved by the crawlers, we will attempt to scrape web pages from the *New York Times* website and extract titles and posts. Then, we evaluate and monitor each crawler individually. We note that *RCrawler* can be directly compared with *Scrapy*, because both are of a similar nature and have similar characteristics. Therefore, both of these obtain the landing page of the website as input, and have the same data extraction rules and configuration parameters. However, *Rvest* has no support

Table 2
Open source Web scraping libraries and frameworks to evaluate.

Project Name	Type	Main role	Language	Parallelism
RCrawler	Library	Crawl & Scrape	R	Yes
Rvest	Library	Retrieve & Parse	R	No
Scrapy	Framework	Crawl & Scrape	Python	Yes

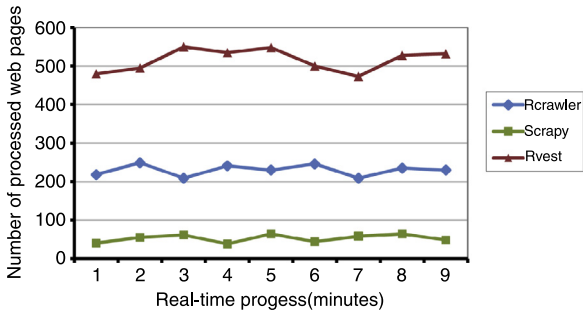

Fig. 7. RCrawler achieves a high performance compared to Rvest, but not as fast as Scrapy.

for crawling or multi-threading. To make a proper comparison, we have implemented a hand-made script of *Rvest*. Here, we provide a list of URLs in advance, and then use a looping function to iterate over them. We apply *Rvest* functions to retrieve and parse web pages, and extract the desired data. We have decided not to consider other R packages mentioned in this paper within the evaluation, because all of them were built around the same basic toolkits as *Rvest* and achieve approximately the same performance.

Fig. 7 shows that *RCrawler* performs better than other R packages in terms of speed. In fact, multi-threading enables efficient use of available resources. However, our crawler is still not efficient as the Python scraper, because of R's shortcomings in data processing. Although the crawler speed is considered as a performance factor, a crawler should also be polite to web servers, such that it does not overload the server with frequent requests within a short space of time. This is especially true for domain-focused crawlers (such as ours). Therefore, there should be sufficient delays between consecutive requests to the same server.

7.3. Evaluating RCrawler scraping

We gathered HTML data in order to analyze the completeness. We used the XPath extraction technique, because this is the only one to be supported in RCrawler so far. Furthermore, we used eight cores and eight connections. As our scraper implementation requires a URL to work, we used a local domain web server with a predefined number of web pages (1000 pages). Among these, there are 800 content pages and 200 menu pages. Every content web page contains two elements that should be extracted: the title and the post. Therefore, we know in advance that the amount of data that should be extracted amounts to 1600 records. The resulting statistics for the experiment are presented in Table 3. It can be seen that the total amount of extracted data achieved by the compared web scrapers comes to between 99% and 100%. In fact, XPath is considered to be among the most reliable extraction techniques, because it is based directly on an XML tree to locate element nodes, attribute nodes, text nodes, and anything else that can occur in an XML document without automatic learning or artificial intelligence. Thus, in our case the scraping function does not have a direct effect on the crawler performance. However, the coverage quality of a crawler plays a highly important role in its performance. This describes the ability to extract all accessible links of a crawled web

Table 3
Open source web scraping performance on scraping.

Project name	Scrapping rate
RCrawler	99.8%
Rvest	99.8%
Scrapy	99.8%

page. At this point, *Rvest* cannot be compared, because it does not have the ability to follow discovered URLs. Indeed, we believe that *RCrawler* is the best R solution for crawling and scraping data in an R environment. It can crawl, download pages, and even scrape content with high speed and efficiency. Besides its ease of use, *RCrawler* also provides a full solution to handling data collection tasks inside R, with many customizable settings and options, and without the need to use an external web crawler/scraper.

8. Conclusions and future work

In this paper, we have presented RCrawler, an R-based, multi-threaded, flexible, and powerful web crawler that provides a suite of useful functions for web crawling, web scraping, and also potentially link analysis.

The implementation of RCrawler also highlights some weaknesses of the R parallel computing environment, which we have been able to overcome by adapting our architecture to R standards and employing superior algorithms when necessary.

RCrawler is an ongoing project. We intend to improve its performance using low level functions in C++ or Rcpp. We also plan to extend its features and functionalities by implementing focused crawling, update management, and automatic content detection and extraction.

References

[1] Markov Z, Larose DT. Data Mining the Web: Uncovering Patterns in Web Content, Structure, and Usage. John Wiley & Sons; 2007.
[2] da Costa MG, Gong Z. Web structure mining: an introduction. In: Information Acquisition, 2005 IEEE International Conference on. IEEE; 2005. p. 6.
[3] Srivastava J, Cooley R, Deshpande M, Tan PN. Web usage mining: Discovery and applications of usage patterns from web data. ACM SIGKDD Explorations Newsletter 2000;1(2):12–23.
[4] Manning CD, Raghavan P, Schütze H, et al. Introduction to Information Retrieval. Cambridge University Press Cambridge; 2008.
[5] Sajja PS, Akerkar R. Intelligent Technologies for Web Applications. CRC Press; 2012.
[6] Liu B. Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data. Springer Science & Business Media; 2007.
[7] Ceri S, Bozzon A, Brambilla M, Della Valle E, Fraternali P, Quarteroni S. Web Information Retrieval. Springer Science & Business Media; 2013.
[8] Ferrara E, De Meo P, Fiumara G, Baumgartner R. Web data extraction, applications and techniques: A survey. Knowl.-Based Syst. 2014;70:301–23.
[9] Zhao Y. R and Data Mining: Examples and Case Studies. Academic Press; 2012.
[10] Thomas L, Scott C, Patrick M, Karthik R, Christopher G, CRAN Task View: Web Technologies and Services; 2016. https://cran.r-project.org/web/views/WebTechnologies.html. [Accessed 29 February 2016].
[11] Munzert S, Rubba C, Meißner P, Nyhuis D. Automated Data Collection with R: A Practical Guide to Web Scraping and Text Mining. John Wiley & Sons; 2014.
[12] Pang B, Lee L. Opinion mining and sentiment analysis. Found. Trends Inf. Retr. 2008;2(1–2):1–135.
[13] Dai XY, Chen QC, Wang XL, Xu J. Online topic detection and tracking of financial news based on hierarchical clustering. In: Machine Learning and Cybernetics (ICMLC), 2010 International Conference on, vol. 6, IEEE; 2010. p. 3341–6.
[14] Ricci F, Rokach L, Shapira B. Introduction to Recommender Systems Handbook. Springer; 2011.
[15] Olston C, Najork M. Web crawling. Foundations and Trends in Information Retrieval 2010;4(3):175–246.
[16] Heydon A, Najork M. Mercator: A scalable, extensible web crawler. World Wide Web 1999;2(4):219–29.
[17] Boldi P, Codenotti B, Santini M, Vigna S. Ubicrawler: A scalable fully distributed web crawler. Softw. - Pract. Exp. 2004;34(8):711–26.
[18] Castillo C. Effective web crawling. In: ACM SIGIR Forum, vol. 39, ACM; 2005. p. 55–6.
[19] Eddelbuettel D. Cran task view: High-performance and parallel computing with r; 2016.
[20] R-core. Package 'parallel'; 2015. https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf. [Accessed 30 February 2016].
[21] Rich C, Revolution Analytics, Steve W, Dan T. doParallel Foreach Parallel Adaptor for the 'parallel' Package; 2015. http://CRAN.R-project.org/package=doParallel. [Accessed 30 February 2016].
[22] Rich C, Analytics R, Steve W. foreach: Provides Foreach Looping Construct for R; 2015. http://CRAN.R-project.org/package=foreach. [Accessed 30 February 2016].
[23] Hadley W. RStudio. httr: Tools for Working with URLs and HTTP; 2016. http://CRAN.R-project.org/package=httr. [Accessed 30 February 2016].
[24] Hadley W, Jeroen O, RStudio, R Foundation. xml2: Parse XML; 2015. http://CRAN.R-project.org/package=xml2. [Accessed 30 February 2016].
[25] Le Hégaret P, Whitmer R, Wood L. Document object model (dom). w3c recommendation; 2005. http://www.w3.org/DOM. [Accessed 30 February 2016].
[26] Lee SH, Kim SJ, Hong SH. On url normalization. In: Computational Science and Its Applications–ICCSA 2005. Springer; 2005. p. 1076–85.
[27] Manku GS, Jain A, Das Sarma A. Detecting near-duplicates for web crawling. In: Proceedings of the 16th International Conference on World Wide Web. ACM; 2007. p. 141–50.
[28] Rabin MO, et al. Fingerprinting by random polynomials Center for Research in Computing Techn., Aiken Computation Laboratory, Univ.; 1981.
[29] Charikar MS. Similarity estimation techniques from rounding algorithms. In: Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing. ACM; 2002. p. 380–8.
[30] Simon U. rJava: Low-Level R to Java Interface; 2016 http://CRAN.R-project.org/package=rJava. [Accessed 30 February 2016].
[31] Dowle M, Srinivasan A, Short T, Lianoglou S, Saporta R, Antonyan E. data.table: Extension of Data.frame; 2015 http://CRAN.R-project.org/package=data.table. [Accessed 30 February 2016].