International Journal on Artificial Intelligence Tools
Vol. 24, No. 2 (2015) 1540005 (22 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218213015400059

A Scalable Approach to Harvest Modern Weblogs

Vangelis Banos
Department of Informatics, Aristotle University of Thessaloniki (AUTH)
Thessaloniki, 54124, Greece
vbanos@gmail.com

Olivier Blanvillain
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
olivier.blanvillain@epfl.ch

Nikos Kasioumis
European Organization for Nuclear Research (CERN)
Geneva 23, 1211, Switzerland
nikos.kasioumis@cern.ch

Yannis Manolopoulos
Department of Informatics, Aristotle University of Thessaloniki (AUTH)
Thessaloniki, 54124, Greece
manolopo@csd.auth.gr

Received 30 September 2014
Accepted 25 November 2014
Published 13 April 2015

Blogs are one of the most prominent means of communication on the web. Their content, interconnections and influence constitute a unique socio-technical artefact of our times which needs to be preserved. The BlogForever project has established best practices and developed an innovative system to harvest, preserve, manage and reuse blog content. This paper presents the latest developments of the blog crawler which is a key component of the BlogForever platform. More precisely, our work concentrates on techniques to automatically extract content such as articles, authors, dates and comments from blog posts. To achieve this goal, we introduce a simple yet robust and scalable algorithm to generate extraction rules based on string matching using the blog's web feed in conjunction with blog hypertext. Furthermore, we present a system architecture which is characterised by efficiency, modularity, scalability and interoperability with third-party systems. Finally, we conduct thorough evaluations of the performance and accuracy of our system.

Keywords: Blog crawler; web data extraction; wrapper generation; interoperability.

1. Introduction

The blogosphere is an established channel of online communication which bears great significance.[17] Wordpress, a single blog publishing company, reports more than 1 million new posts and 1.5 million new comments each day.[38] These overwhelming numbers illustrate the importance of blogs in most aspects of private and business life.[5] Blogs contain data with historic, political, social and scientific value which need to be accessible for current and future generations. For instance, blogs proved to be an important resource during the 2011 Egyptian revolution by playing an instrumental role in the organization and implementation of protests.[8] The problem is that blogs disappear every day[16] because there is no standard method or authority to ensure blog archiving and long-term digital preservation.

Among the challenges in developing a blog archiving software is the design of a web crawler capable of efficiently traversing blogs to harvest their content. The sheer size of the blogosphere combined with an unpredictable publishing rate of new information call for a highly scalable system, while the lack of programmatic access to the complete blog content makes the use of automatic extraction techniques necessary. The variety of available blog publishing platforms offers a limited common set of properties that a crawler can exploit, further narrowed by the ever-changing structure of blog contents. Finally, an increasing number of blogs heavily rely on dynamically created content to present information, using the latest web technologies, hence invalidating traditional web crawling techniques.

A key characteristic of blogs which differentiates them from regular websites is their association with web feeds.[23] Their primary use is to provide a uniform subscription mechanism, thereby allowing users to keep track of the latest updates without the need to actually visit blogs. Concretely, a web feed is an XML file containing links to the latest blog posts along with their articles (abstract or full text) and associated metadata.[31] While web feeds essentially solve the question of update monitoring, their limited size makes it necessary to download blog pages to harvest previous content.

This paper presents the latest developments of the open-source BlogForever Crawler, a key component of the BlogForever platform[18] responsible for traversing blogs, extracting their content and monitoring their updates. Our main objectives in this work are to introduce a new approach to blog data extraction and to present the architecture and implementation of a blog crawler capable of extracting articles, authors, publication dates, comments and potentially any other element which appear in weblog web feeds. Our contributions can be summarized as follows:

- A new algorithm to build extraction rules from web feeds and an optimised reformulation based on a particular string similarity algorithm featuring linear time complexity.
- A methodology to use the algorithm for blog article extraction and how it can be augmented to be used with other blog elements such as authors, publication dates and comments.

- The overall BlogForever crawler architecture and implementation with a focus on design decisions, modularity, scalability and interoperability.
- An approach to use a complete web browser to render JavaScript powered web pages before processing them. This step allows our crawler to effectively harvest blogs built with modern technologies, such as the increasingly popular third-party commenting systems.
- A mapping of the extracted blog content to Archival Information Packages (AIPs) using METS and MARCXML standards for interoperability purposes.
- An evaluation of the content extraction and execution time of our algorithm against three state-of-the-art web article extraction algorithms.

The concepts emerging from our research are viewed in the context of the BlogForever platform but the presented algorithms, techniques and system architectures can be used in other applications related to Wrapper Generation and Web Data Extraction.

The rest of this work is structured as follows: Section 2 presents related work. Section 3 introduces the new algorithms to extract data from blogs. Section 4 presents the blog crawler system architecture and implementation. Section 5 presents the evaluation and results. Finally, our conclusions and some discussion on our work are presented in section 6.

2. Related Work

Web crawlers are complex software systems which often combine techniques from various disciplines in computer science. Our work on the BlogForever crawler is related to the fields of web data extraction, distributed computing and natural language processing. In the literature on web data extraction, the word *wrapper* is commonly used to designate procedures to extract structured data from unstructured documents. We did not use this word in the present paper in favour of the term *extraction rule*, which better reflects our implementation and is decoupled from software that concretely performs the extraction.

A common approach in web data extraction is to manually build wrappers for the targeted websites. This approach has been proposed for the crawler discussed in Ref. 9 which automatically assigns web sites to predefined categories and gets the appropriate wrapper from a static knowledge base. The limiting factor in this type of approach is the substantial amount of manual work needed to write and maintain the wrappers, which is not compatible with the increasing size and diversity of the web. Several projects try to simplify this process and provide various degrees of automation. This is the case of the Stalker algorithm[26] which generates wrappers based on user-labelled training examples. Some commercial solutions such as the Lixto project[12] simplify the task of building wrappers by offering a complete integrated development environment where the training data set is obtained via a graphical user interface.

As an alternative to dedicated software for the creation and maintenance of wrappers, some query languages have been designed specifically for wrappers. These languages rely on their users to manually identify the structure of the data to be extracted. This structure can then be formalised as a small declarative program, which can then be turned into an concrete wrapper by an execution engine. The OXPath language[10] is an interesting extension to XPath designed to incorporate interaction in the extraction process. It supports simulated user actions such as filling forms or clicking buttons to obtain information that would not be accessible otherwise. Another extension of XPath, called Spatial XPath,[28] allows to write spacial rules in the extraction queries. The execution engine embeds a complete web browser which computes the visual representation of the page.

Fully automated solutions use different techniques to identify and extract information directly from the structure and content of the web page, without the need of any manual intervention. The Boilerpipe project[19] (which is also used in our evaluation) uses text density analysis to extract the main article of a web page. The approach presented in Ref. 29 is based on a tree structure analysis of pages with similar templates, such as news web sites or blogs. Automatic solutions have also been designed specifically for blogs. Similarly to our approach, Oita and Senellart[27] describe a procedure to automatically build wrappers by matching web feed articles to HTML pages. This work was further extended by Gkotsis *et al.* with a focus on extracting content anterior to the one indexed in web feeds.[11] They also report to have successfully extracted blog post titles, publication dates and authors, but their approach is less generic than the one for the extraction of articles. Finally, neither Ref. 27 nor Ref. 11 provide complexity analysis which we believe to be essential before using an algorithm in production.

One interesting research direction is the one of large scale distributed crawlers. Mercator,[14] UbiCrawler[2] and the crawler discussed in Ref. 32 are examples of successful distributed crawlers. The associated articles provide useful information regarding the challenges encountered when working on a distributed architecture. One of the core issues when scaling out seems to be in sharing the list of URLs that have already been visited and those that need to be visited next. While Refs. 14 and 32 rely on a central node to hold this information,[2] uses a fully distributed architecture where URLs are divided among nodes using consistent hashing. Both of these approaches require the crawlers to implement complex mechanisms to achieve fault tolerance. The BlogForever Crawler does not have to address this issue as it is already Handled by the BlogForever back-end system which is responsible for task and state management. In addition, since we process web pages on the fly and directly emit the extracted content to the back-end, there is no need for persistent storage on the crawler's side. This removes one layer of complexity when compared to general crawlers which need to use a distributed file system (Ref. 32 uses NFS, Ref. 1 uses HDFS) or implement an aggregation mechanism to further exploit the collected data. Our design is similar to the distributed active object pattern presented in Ref. 20, which is further simplified by the fact that the state of the crawler instances is not kept between crawls.

3. Algorithms

We propose a new set of algorithms to extract blog post articles as well as variations for extracting authors, dates and comments. We start with our motivation to use blog specific characteristics, followed by our approach to build *extraction rules* which are applicable throughout a blog. Our focus is on minimising the algorithmic complexity while keeping our approach simple and generic.

3.1. *Motivation*

Extracting metadata and content from HTML documents is a challenging task because standards and format recommendations suffer from very low usage. W3C has been publishing web standards and format recommendations for quite some time.[34] For instance, according to the W3C HTML guidelines, the `<h1></h1>` tags have to contain the highest-level heading of the page and must not appear more than once per page.[35] More recently, specifications such as microdata[36] define ways to embed semantic information and metadata inside HTML documents, but these still suffer from very low usage: estimated to be used in less than 0.5% of websites.[30] In fact, the majority of websites rely on the generic `<span></span>` and `<div></div>` container elements with custom `id` or `class` attributes to organise the structure of pages, and more than 95% of pages do not pass HTML validation.[37] Under such circumstances, relying on HTML structure to extract content from web pages is not viable and other techniques need to be employed.

Having blogs as our target websites, we made the following observations which play a central role in the extraction process:

(1) Blogs provide web feeds: structured and standardized views of the latest posts of a blog.
(2) Posts of the same blog share a similar HTML structure.

Web feeds usually contain about 20 blog posts,[27] often less than the total number of posts in blogs. Consequently, to effectively archive the entire content of a blog, it is necessary to download and process pages beyond the ones referenced in the web feed.

3.2. *Content extraction overview*

To extract content from blog posts, we proceed by building *extraction rules* from the data given in the blog's web feed. The idea is to use a set of *training data*, pairs of HTML pages and target content, which are used to build an extraction rule capable of locating the target content on each HTML page.

Observation (1) allows the crawler to obtain input for the extraction rule generation algorithm: each web feed entry contains a link to the corresponding web page as well as blog post article (either abstract or full text), its title, authors and publication date. We call these fields *targets* as they constitute the data our crawler aims to extract. Observation (2) guarantees the existence of an appropriate extraction rule, as well as its applicability to all posts of the blog. Each page is uniquely identified by its URL. If a page is already processed, it is not processed again in the future.

Algorithm 1 shows the generic procedure we use to build extraction rules. The idea is quite simple: for each (*page, target*) input, compute, out of all possible extraction rules, the best one with respect to a certain `ScoreFunction`. The rule which is most frequently the *best rule* is then returned.

Algorithm 1: Best Extraction Rule

```
input  : Set pageZipTarget of (page and target) pairs
output: Best extraction rule

bestRules ⟵ new list
foreach (page, target) in pageZipTarget do
    score ⟵ new map
    foreach rule in AllRules(page) do
        extracted ⟵ Apply(rule, page)
        score of rule ⟵ ScoreFunction(extracted, target)
    bestRules ⟵ bestRules + rule with highest score
return rule with highest occurrence in bestRules
```

One might notice that each *best rule* computation is independent and operates on a different input pair. This implies that Algorithm 1 is *embarrassingly parallel*: iterations of the outer loop can trivially be executed on multiple threads.

Functions in Algorithm 1 are voluntarily abstract at this point and will be explained in detail in the remaining of this section. Subsection 3.3 defines `AllRules`, `Apply` and the `ScoreFunction` we use for article extraction. In subsection 3.4 we analyse the time complexity of Algorithm 1 and give a linear time reformulation using dynamic programming. Finally, subsection 3.5 shows how the `ScoreFunction` can be adapted to extract authors, dates and comments.

3.3. *Extraction rules and string similarity*

In our implementation, rules are queries in the XML Path Language (XPath). Consequently, standard libraries can be used to parse HTML pages and apply extraction rules, providing the `Apply` function used in Algorithm 1. We experimented with 3 types of XPath queries: selection over the HTML `id` attribute, selection over the HTML `class` attribute and selection using the relative path in the HTML tree. `id` attributes are expected to be unique, and `class` attributes have showed in our experiments to have better consistency than relative paths over pages of a blog. For these reasons we opt to always favour `class` over path, and `id` over `class`, such that the `AllRules` function returns a single rule per node.

Function `AllRules`(*page*)

```
rules ⟵ new set
foreach node in page do
    if node as id attribute then
        rules ⟵ rules + {"//*[@id='node.id']"}
    else if node as class attribute then
        rules ⟵ rules + {"//*[@class='node.class']"}
    else rules ⟵ rules + {RelativePathTo(node)}
return rules
```

Unsurprisingly, the choice of `ScoreFunction` greatly influences the running time and precision of the extraction process. When targeting articles, extraction rule scores are computed with a string similarity function comparing the extracted strings with the target strings. We chose the Sorensen–Dice coefficient similarity,[6] which is, to the best of our knowledge, the only string similarity algorithm fulfilling the following criteria:

(1) Has low sensitivity to word ordering,
(2) Has low sensitivity to length variations,
(3) Runs in linear time.

Properties 1 and 2 are essential when dealing with cases where the blog's web feed only contains an abstract or a subset of the entire post article. Table 1 gives examples to illustrate how these two properties hold for the Sorensen–Dice coefficient similarity but do not for *edit distance* based similarities such as the Levenshtein[22] similarity.

Table 1. Examples of string similarities.

String1	String2	Sorensen–Dice	Levenshtein
`"Scheme Scala"`	`"Scala Scheme"`	90%	50%
`"Rachid"`	`"Richard"`	18%	61%
`"Rachid"`	`"Amy, Rachid and all their friends"`	29%	31%

The Sorensen–Dice coefficient similarity algorithm operates by first building sets of pairs of adjacent characters, also known as *bigrams*, and then applying the *quotient of similarity* formula:

```
Function Similarity(string1, string2)
bigrams1 ⟵ Bigrams(string1)
bigrams2 ⟵ Bigrams(string2)
return 2 |bigrams1 ∩ bigrams2| / (|bigrams1| + |bigrams2|)
```

```
Function Bigrams(string)
return set of pairs of adjacent characters in string
```

3.4. *Time complexity and linear reformulation*

With the functions `AllRules`, `Apply` and `Similarity` (as `ScoreFunction`) being defined, the definition of Algorithm 1 for article extraction is now complete. We can therefore proceed with a time complexity analysis.

First, let us assume that we have at our disposal a linear time HTML parser that constructs an appropriate data structure, indexing HTML nodes on their `id` and `class` attributes, effectively making `Apply` $\in \mathcal{O}(1)$. As stated before, the outer loop splits the input into independent computations and each call to `AllRules` returns (in linear time) at most as many rules as the number of nodes in its *page* argument. Therefore, the body of the inner loop will be executed $\mathcal{O}(n)$ times. Because each extraction rule can return any subtree of the queried page, each call to `Similarity` takes $\mathcal{O}(n)$, leading to an overall quadratic running time.

We now present Algorithm 2, a linear time reformulation of Algorithm 1 for article extraction using dynamic programming.

While very intuitive, the original idea of first generating extraction rules and then picking these best rules prevents us from effectively reusing previously computed N-grams (sets of adjacent characters). For instance, when evaluating the extraction rule for the HTML root node, Algorithm 1 will obtain the complete string of the page and pass it to the `Similarity` function. At this point, the information on where the string could be split into substrings with already computed N-grams is not accessible, and the N-grams of the page have to be computed by linearly traversing the entire string. To overcome this limitation and implement *memoization* over the N-gram computations, Algorithm 2 uses a post-order traversal of the HTML tree and computes node N-grams from their children N-grams. This way, we avoid serializing HTML subtrees for each N-gram computation and have the guarantee that each character of the HTML page will be read at most once during the N-gram computation.

An interesting question is what is the optimal N for N-gram computation. To answer this question, we conduct some simple experiments. Using sample blog post

Algorithm 2: Linear Time Best Content Extraction Rule

```
input  : Set pageZipTarget of (Html and Text) pairs
output: Best extraction rule

bestRules ⟵ new list
foreach (page, target) in pageZipTarget do
    score ⟵ new map
    bigrams ⟵ new map
    bigrams of target ⟵ Bigrams(target)
    foreach node in page with post-order traversal do
        bigrams of node ⟵
            Bigrams(node.text) ∪ bigrams of all node.childs
        score of node ⟵
            $\frac{2\ |(\textit{bigrams}\ \textbf{of}\ \textit{node}) \cap (\textit{bigrams}\ \textbf{of}\ \textit{target})|}{|\textit{bigrams}\ \textbf{of}\ \textit{node}| + |\textit{bigrams}\ \textbf{of}\ \textit{target}|}$
    bestRules ⟵ bestRules + Rule(node with best score)
return rule with highest occurrence in bestRules
```

excerpts and full texts, we calculate the string similarity between each excerpt and full text. For instance, using a blog post full text and associated text excerpt from the popular blog TechCrunch as presented in Table 2, the text similarity results (Table 3) using different N values support the selection of bigrams ($N = 2$).

With bigrams computed in this dynamic programming manner, the overall time to compute all `Bigrams(`*node.text*`)` is linear. To conclude the argument that Algorithm 2 runs in linear time we show that all other computations of the inner loop can be done in constant *amortized* time. As the number of edges in a tree is one less than the number of nodes, the *amortized* number of bigrams unions per inner loop iteration tends to one. Each *quotient of similarity* computation requires one bigrams intersection and three bigrams length computations. Over a finite alphabet (we used printable ASCII), bigrams sizes have bounded size and each of these operations takes constant time.

3.5. *Variations for authors, dates, comments*

Using string similarity as the only score measurement leads to poor performance on author and date extraction, and is not suitable for comment extraction. This subsection presents variations of the `ScoreFunction` which addresses issues of these other types of content.

The case of authors is problematic because authors' names often appear in multiple places of a page, which results in several rules with maximum `Similarity` score. The heuristic we use to get around this issue consists of adding a new

Table 2. TechCrunch blog post example.

Blog Post Text
Apple has a new version of iOS 8 out, just a short time after the initial launch of the software. The update, 8.0.1, includes a number of fixes, but most notably (and listed first), it addressed the bug that prevented HealthKit apps from being available at launch. It also zaps some bugs with third-party keyboards, which should make them remain the default option until a user switches to another, which has been a sore spot for fans of the new external software keyboard options. Unfortunately, installing the iOS 8.0.1 update revealed that despite Apple's promised fixes, it actually completely disables cellular service and Touch ID on many devices, though some iPhone 5s and older model owners report no issues. The bottom line is that you should definitely NOT install this update, at least until an updated version appears, at which time we will let you know it is safe to go ahead. As you can see in the image below, Apple is also addressing an issue that blocked some photos from appearing in Photo Library, fixing reliability concerns around Reachability on iPhone 6 and 6 Plus (which brings the top of the screen down when you double touch the Home button), zapping bugs that cause unexpected data use when communicating via SMS or MMS, and improving the "Ask to Buy" feature for Family Sharing, specifically around in-app purchases, in addition to other minor bugs.
Weblog Post Text Abstract from RSS
Apple has a new version of iOS 8 out, just a short time after the initial launch of the software. The update, 8.0.1, includes a number of fixes, but most notably (and listed first), it addressed the bug that prevented HealthKit apps from being available at launch. It also zaps ...

Table 3. Blog post excerpt and full text similarity using different N values.

N	2	3	4
Similarity	0.7817	0.6680	0.6212

component in the `ScoreFunction` for author extraction rules: the *tree distance* between the evaluated node and the post content node. This new component takes advantage of the positioning of a post's authors node which often is a direct child or shares its parent with the post content node.

Dates are affected by the same duplication issue, as well as the issue of inconsistencies of format between web feeds and web pages. Our solution for date extraction extends the `ScoreFunction` for authors by comparing the *extracted* string to multiple *targets*, each being a different string representation of the original date obtained from the web feed. For instance, if the feed indicates that a post was published at `"Thu, 01 Jan 1970 00:00:00"`, our algorithm will search for a rule that returns one of `"Thursday January 1, 1970"`, `"1970-01-01"`, `"43 years ago"` and so on. So far we do not support dates in multiple languages, but adding new target formats based on languages detection would be a simple extension of our date extraction algorithm.

Comments are usually available in separate web feeds, one per blog post. Similarly to blog feeds, comment feeds have a limited number of entries, and when the number of comments on a blog post exceeds this limit, comments have to be extracted from web pages. To do so, we use the following `ScoreFunction`:

- Rules returning fewer HTML nodes than the number of comments on the feed are filtered out with a zero score,
- The scores of the remaining rules are computed with the value of the *maximum weighted matching* in the *complete bipartite graph* $G = (U, V, E)$, where U is the set of HTML nodes returned by the rule, V is the set of target comment fields from the web feed (such as comment authors) and $E(u, v)$ has weight equal to `Similarity(`u, v`)`.

Regarding time complexity, computing the *tree distance* of each node of a graph to a single reference node and multiplying the number of targets by a constant factor can be done in linear time. However, computing scores of comment extraction rules requires a more expensive algorithm. This is compensated by the fact that the proportion of candidate HTML nodes left, after filtering out rules not returning enough results, is very low in practice. Analogous reformulations to the one done with Algorithm 2 can be straightforwardly applied on each ScoreFunction to minimize the time spent in `Similarity` calculations.

It must be noted that there is no limitation due to comment nesting as long as comments follow the same format.

4. Architecture

We present the BlogForever crawler system architecture which implements the proposed algorithms for weblog data extraction via the generation of extraction rules. We describe the system architecture and discuss the software tools and techniques we used, such as the enrichment of the Scrapy framework for our specific usage and the integration of a headless web browser into the harvesting process to achieve content extraction from webpages which use JavaScript to display content. Following, we focus on the scalability design and distributed architecture of our system. Finally, we present our provisions for interoperability using established open standards which increases the value and reusability of the proposed system in many contexts.

4.1. *System and workflow*

The BlogForever crawler is a Python[a] application which is based on Scrapy, an open-source framework for web crawling. Scrapy provides an elegant and modular architecture illustrated in Fig. 1. Several components can be plugged into the Scrapy core infrastructure. Following, we present each part of the architecture and our own contributions:

[a]`http://www.python.org/`

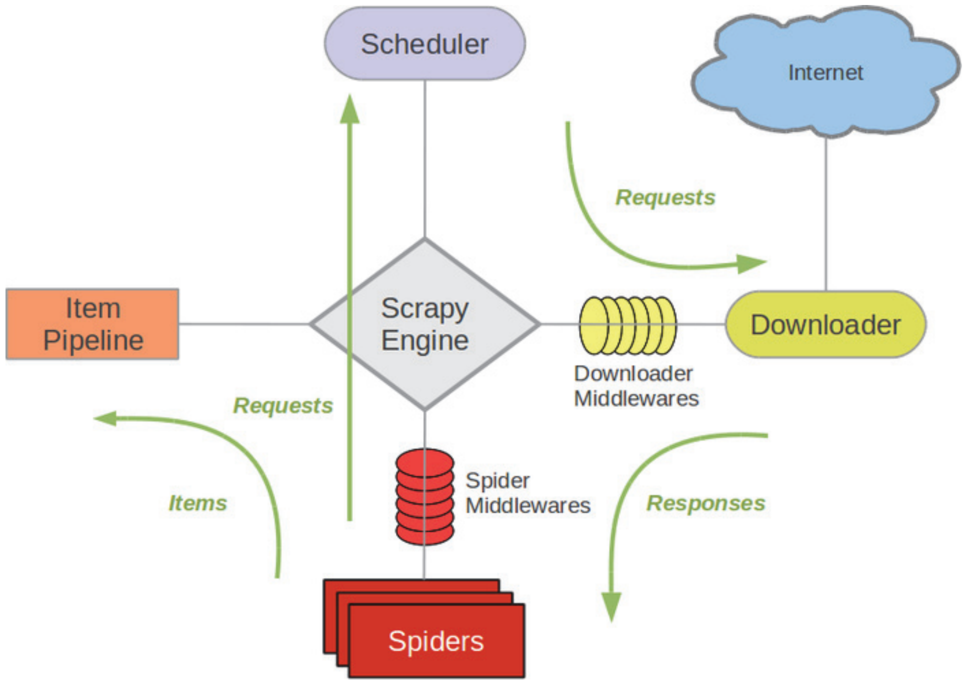

Fig. 1. Overview of the crawler architecture. (Credit: Pablo Hoffman, Daniel Graña, Scrapy)

- *Spiders* define how a target website is scraped, including how to perform the crawl (i.e. follow links). The BlogForever crawer implementation includes two new types of spiders: *NewCrawl* and *UpdateCrawl*, which implement the logic to respectively crawl a new blog and get updates from a previously crawled blog.
- *Item Pipeline* defines the processing of extracted data from the spiders through several components that are executed sequentially. The BlogForever crawler implementation includes a new item pipeline which orchestrates all aspects of crawling. More specifically the BlogForever pipeline is defined as follows:

  (1) JavaScript rendering,
  (2) Extract content,
  (3) Extract comments,
  (4) Download multimedia files,
  (5) Prepare Archival Information Packages (APIs) to propagate the results to potential back-ends.

- *Downloader Middlewares* is a framework of hooks into Scrapy's request/response processing and altering Scrapy's requests and responses.
- *Spider Middlewares* is a framework of hooks into Scrapy's spider processing mechanism.

The system architecture providers great modularity. This is illustrated clearly in our work with the following example:

- If it is necessary to disable JavaScript rendering or plugging in an alternative back-end can be done by editing a single line of code.
- The features to extract comments and download multimedia files were implemented after creating the initial logic to extract content and were added as extra steps in the pipeline.
- The requirement to implement interoperability provisions later presented in section 4.4 was easily covered with the implementation of an extra middleware plugin which was invoked from the main crawler architecture. No further modifications were necessary in the code.

In the remaining parts of this section, we elaborate our work on each specific part of the crawler system.

4.2. *JavaScript rendering*

JavaScript is a widely used language for client-side scripting. While some applications simply use it for aesthetics, an increasing number of websites use JavaScript to download and display content. In such cases, traditional HTML based crawlers do not see web pages as they are presented to a human visitor by a web browser, and might therefore be obsolete for data extraction.

In our experiments whilst crawling the blogosphere, we encountered several blogs where crawled data was incomplete because of the lack of JavaScript interpretation. The most frequent cases were blogs using the Disqus[b] and LiveFyre[c] comment hosting services. For webmasters, these tools are very handy because the entire comments infrastructure is externalized and their setup essentially comes down to including a JavaScript snippet in each target page. Both of these services heavily rely on JavaScript to download and display the comments, even providing functionalities such as real-time updates for edits and newly written comments. Less commonly, some blogs are fully rendered using JavaScript. When loading such websites, the web browser will not receive the page content as an HTML document, but will instead have to execute JavaScript code to download and display the page content. The Blogger platform provides the *Dynamic Views* as a default template, which uses this mechanism.[13]

To support blogs with JavaScript-generated content, we embed a full web browser into the crawler. After considering multiple options, we opted for PhantomJS,[d] a headless web browser with great performance and scripting capabilities. The JavaScript rendering is enabled by default and is the very first step

[b]`http://disqus.com/websites`
[c]`http://web.livefyre.com`
[d]`http://phantomjs.org`

of web page processing. Therefore, extracting blog post articles, comments or multimedia files works equally well on blogs with JavaScript-generated content and on traditional HTML-only blogs.

When the number of comments on a page exceeds a certain threshold, both Disqus and LiveFyre will only load the most recent ones and the stream of comments will end with a *Show More Comments* button. As part of the page loading process, we instruct PhantomJS to repeatedly click on these buttons until all comments are loaded. Paths to Disqus and LiveFyre *Show More* buttons are manually obtained. They constitute the only non-generic elements of our extraction stack which require human intervention to maintain and extend to other commenting platforms.

4.3. *Content extraction*

In order to identify web pages as blog posts, our implementation enriches Scrapy with two components to narrow the extraction process down to the subsets of pages which are blog posts: *blog post identification* and *download priority heuristic*.

Given a URL entry point to a website, the default Scrapy behaviour traverses all the pages of the same domain in a *last-in-first-out* manner. The *blog post identification* function is able to identify whether an URL points to a blog post or not. Internally, for each blog, this function uses a regular expression constructed from the blog post URLs found in the web feed. This simple approach requires that blogs use the same URL pattern for all their posts (or false negatives will occur) which has to be distinct for pages that are not posts (or false positives will occur). In practice, this assumption holds for all blog platforms we encountered and seems to be a common practice among web developers.

In order to efficiently deal with blogs that have a large number of pages which are not posts, the *blog post identification* mechanism is not sufficient. Indeed, after all pages identified as blog posts are processed, the crawler needs to download all other pages to search for additional blog posts. To replace the naive *random walk*, *depth first search* or *breadth first search* web site traversals, we use a priority queue where priorities for new URLs are determined by a machine learning system. This mechanism has shown to be mandatory for blogs hosted on a single domain alongside large number of other types of web pages, such as those in forums or wikis.

The idea is to give high priority to URLs which are believed to point to pages with links to blog posts. These predictions are done using an active *Distance-Weighted k-Nearest-Neighbour* classifier.[7] Let $L(u)$ be the number of links to blog posts contained in a page with URL u. Whenever a page is downloaded, its URL u and $L(u)$ are given to the machine learning system as training data. When the crawler encounters a new URL v, it will ask the machine learning system for an estimation of $L(v)$, and use this value as the download priority of v. $L(v)$ is estimated by calculating a weighted average of the values of the k URLs most similar to v.

4.4. *The BlogForever metadata schema for interoperability*

One of the key BlogForever project goals is interoperability with third party platforms. The original BlogForever crawler was intended to insert blog data directly to the BlogForever repository component but later the architecture was reworked to make it possible to use other storage and archiving systems as well. To achieve this goal, we implement a special interoperability middleware for the spider to produce Archival Information Packages (AIPs) from harvested blog content. The AIPs can be used by any software platform which complies with the OAIS reference model.[21] It must be noted also that this is the first time weblog content is encoded in this way.

The AIPs consist of XML files structured using the METS[4] standard for encoding metadata and content. METS is widely adopted and supported by all popular digital library systems. In addition, the blog content attributes which are included in the METS XML packages are encoded using the MARCXML Schema.[25] The reason for the selection of MARCXML is the wide adoption of the standard, its flexibility and extensibility, as well as previous experience with the Invenio digital library system[e] which is also based on MARCXML.

There are three kinds of entities which can be included in an AIP: Blog, Entry and Comment. The content extracted from weblogs is mapped to the relevant entities using the following rule: If an attribute is already defined in MARC for other content types use the same MARC code for blogs. If an attribute is totally new, an unused MARC 9xx tag is chosen to represent it, composing therefore the BlogForever metadata schema.[24] Following, we present the BlogForever metadata schema for Blog, Post, Page and Comment entities in Tables 4–6.

4.5. *Distributed architecture and scalability*

One of the problems of web crawling is the large amount of input which need to be processed. To address this issue, it is crucial to build every layer of the system with scalability in mind.[33]

The BlogForever Crawler, and in particular the two core procedures *NewCrawl* and *UpdateCrawl*, are designed to be usable as part of an event-driven, scalable and fault-resilient distributed system. Heading in this direction, we made the key design choice to have both *NewCrawl* and *UpdateCrawl* as stateless components. From a high-level point of view, these two components are *purely functional*:

$$\begin{aligned} \mathit{NewCrawl} &: \ \mathit{URL} \rightarrow \mathcal{P}(\mathit{RECORD}) \\ \mathit{UpdateCrawl} &: \ \mathit{URL} \times \mathit{DATE} \rightarrow \mathcal{P}(\mathit{RECORD}) \end{aligned}$$

where *URL*, *DATE* and *RECORD* are respectively the set of all URLs, dates and records, and $\mathcal{P}$ designates the power set operator. By delegating all shared mutable state to the back-end system, web crawler instances can be added, removed and used interchangeably.

[e]`http://invenio-software.org/`

Table 4. Blog record attributes — MARC 21 representations mapping.

Blog Attribute	MARC 21 Representation
title	245 $a
subtitle	245 $b
URI	520 $u
aliases	100 $g
status_code	952 $a
language	041 $a
encoding	532
sitemap_uri	520
platform	781 $a
platform_version	781 $b
webmaster	955 $a
hosting_ip	956 $a
location_city	270 $d
location_country	270 $b
last_activity_date	954 $a
post_frequency	954 $b
update_frequency	954 $c
copyright	542
ownership_rights	542
distribution_rights	542
access_rights	542
license	542 $f

Table 5. Blog record attributes — MARC 21 representations mapping.

Post and Page Attribute	MARC 21 Representation
title	245 $a
subtitle	245 $b
full_content	520 $a
full_content_format	520 $b
author	100 $a
URI	520 $u
aliases	100 $g
alt_identifier (UR)	0247 $a
date_created	269 $c
date_modified	260 $m
version	950 $a
status_code	952 $a
response_code	952 $b
geo_longitude	342 $g
geo_latitude	342 $h
access_restriction	506
has_reply	788 $a
last_reply_date	788 $c
num_of_replies	788 $b
child_of	760 $o $4 $w

Table 6. Comment record attributes — MARC tags mapping.

Comment Attribute	MARC 21 Representation
subject	245 $a
author	100 $a
full_content	520 $a
full_content_format	520 $b
URI	520 $u
status	952 $a
date_added	269 $c
date_modified	269 $m
addressed_to_URI	789 $u
geo_longitude	342 $g
geo_latitude	342 $h
has_reply	788 $a
num_replies	788 $b
is_child_of_post	773 $o $4 $w
is_child_of_comment	773 $o $4 $w

To implement a distributed crawler architecture, we choose to use Scrapyd,[f] an application for deploying and running Scrapy spiders. The process is quite straightforward:

(1) Deploy the BlogForever crawler in any number of servers according to requirements. Using the Scrapyd component which is run as a system daemon, each crawler is listening for requests to run crawling tasks and spawn a process for each new command.
(2) Implement a small control program that reads the list of target weblogs which need to be crawled and issue commands in a round-robin fashion using the Scrapyd JSON API.[g]
(3) All crawlers share a common storage service where they save the crawling results.

5. Evaluation

Our evaluation is articulated in two parts. First, we compare the article extraction procedure presented in section 3 with three open-source projects capable of extracting articles and titles from web pages. The comparison will show that our weblog-targeted solution has better performance both in terms of success rate and running time. Second, a discussion is held regarding the different solutions available to archive data beyond what is available in the HTML source code. Extraction of authors, dates and comments is not part of this evaluation because of the lack of publicly available competing projects and reference data sets.

In our experiments we used *Debian GNU/Linux 7.2*, *Python 2.7* and an *Intel Core i7-3770 3.4 GHz* processor. Timing measurements were made on a single

[f]`http://scrapyd.readthedocs.org/en/latest/`
[g]`http://scrapyd.readthedocs.org/en/latest/api.html`

Table 7. Extraction success rates for different algorithms.

Target	Our Approach	Readability	Boilerpipe	Goose
Article	93.0%	88.1%	79.3%	79.2%
Title	95.0%	74.0%	N/A	84.9%

dedicated core with garbage collection disabled. The Git repository for this paper[h] contains the necessary scripts and instructions to reproduce all the evaluation experiments presented in this section. The crawler source code is available under the MIT license from the project's websites.[i]

5.1. *Extraction success rates*

To evaluate article and title extraction from weblog posts we compare our approach to three open source projects: Readability,[j] Boilerpipe[19] and Goose,[k] which are implemented in JavaScript, Java and Scala respectively. These projects are more generic than our blog-specific approach in the sense that they are able to identify and extract data directly from HTML source code, and do not make use of web feeds or structural similarities between pages of the same weblog (observations (1) and (2)). Table 7 shows the extraction success rates for article and title on a test sample of 2300 posts from 230 weblogs obtained from the Spinn3r dataset.[3]

On our test dataset, Algorithm 1 outperformed the competition by 4.9% on article extraction and 10.1% on title extraction. It is important to stress that Readability, Boilerpipe and Goose rely on generic techniques such as word density, paragraph clustering and heuristics on HTML tagging conventions, which are designed to work for any type of web page. On the contrary, our algorithm is only suitable for pages with associated web feeds, as these provide the reference data used to build extraction rules. Therefore, results shown in Table 7 should not be interpreted as a general quality evaluation of the different projects, but simply as evidence that our approach is more suitable when working with weblogs.

5.2. *Article extraction running times*

In addition to the quality of the extracted data we also evaluated the running time of the extraction procedure. The main point of interest is the ability of the extraction procedure to scale as the number of posts in the processed weblog increases. This corresponds to the evaluation of a *NewCrawl* task, which is in charge of harvesting all published content on a weblog.

Figure 2 shows the cumulated time spent for each article extraction procedure (this excludes common tasks such as downloading pages and storing results) as a

[h] `https://github.com/OlivierBlanvillain/bfc-paper`
[i] `https://github.com/BlogForever/crawler`
[j] `https://github.com/gfxmonk/python-readability`
[k] `https://github.com/GravityLabs/goose`

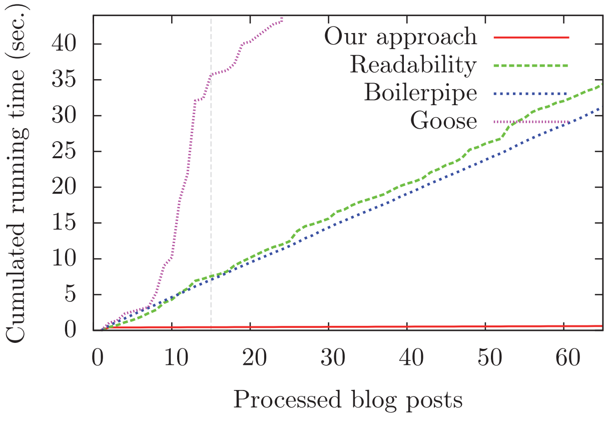

Fig. 2. Running time of articles extraction.

function of the number of weblog posts processed. We used the Quantum Diaries[1] blog for this experiment. Data presented in this graph was obtained by taking the arithmetic mean over 10 measurements. These results are believed to be significant given that standard deviations are of the order of 2 milliseconds.

As illustrated in Figure 2, our approach spends the majority of its total running time between the initialisation the processing of the first weblog post. This initial increase of about 0.4 seconds corresponds to cost of executing Algorithm 2 to compute extraction rule for articles. As already mentioned, this consists in computing the *best extraction rule* of each pages references by the web feed and picking the one functioning best on these pages. Once we have this extraction rule, processing subsequent weblog posts only requires parsing and applying the rule, which takes about 3 milliseconds and are barely visible on the scale of Figure 2. The other evaluated solutions do not function this way: each weblog post is processed as new and independent input, leading to approximately linear running times.

The vertical dashed line at 15 processed weblog posts represents a suitable point of comparison of processing time per post. Indeed, as the web feed of our test weblog contains 15 posts, the extraction rule computation performed by our approach include the cost of entirely processing these 15 entries. That being said, comparing raw performance of algorithms implemented in different programming languages is not very informative given the high variations of running times observed across programming languages.[15]

When compared to the other parts of crawling, the content extraction is sufficiently quick not to be a bottleneck. Indeed, extracting the content of 100 pages takes about half a second, downloading 100 pages takes at least 1 second, and, if enabled, rendering and taking screenshots takes about half a second per page.

[1]`http://www.quantumdiaries.org`

6. Discussion and Conclusions

In this article, we presented our extended work on a scalable approach to harvest modern weblogs. Our approach is based on a new algorithm to build extraction rules from web feeds. The key observations which led us to the inception and implementation of this algorithm are the facts that: (1) weblogs provide web feeds which include structured and standardized views of the latest blog posts, and (2) post of the same weblog share a similar HTML structure. Following, we presented a simple adaptation of this procedure that allows extracting different types of content, including authors, dates, comments and potentially any other element. The elaboration of this process enables the wider use of this method as it is feasible to devise variations of the proposed method to extract any kind of weblog content.

A critical part of this work was the presentation of the BlogForever crawler architecture and discussion of the software tools and the novel techniques we used. In order to support rapidly evolving web technologies such as JavaScript-generated content, the crawler uses a headless web browser to render pages before processing them. Another important aspect of the crawler architecture was the interoperability provisions. In our design, we introduced a new metadata schema for interoperability, encoding weblog crawling results in Archival Information Packages (AIPs) using established open standards. This feature enables the use of our system in many contexts and with multiple different back-ends, increasing its relevance and reusability. We also highlighted the design choices made to achieve both modularity and scalability. This is particularly relevant in the domain of web crawling given that intensive network operations can be a serious bottleneck. Crawlers greatly benefit from the use of multiple Internet access points which makes them natural candidates for distributed computing.

Our method had great success with content extraction accuracy and performance against state-of-the-art open source web article extraction systems as presented in the evaluation section. We have to note thought that our experiments on a considerably large weblogs dataset showed that there were some failing tests which stem from either the violation of one of our two key observations, or from an insufficient amount of text in posts. Therefore, it is suggested to potential users to ensure that these observations are valid on the target weblogs before proceeding with using the BlogForever crawler. Future work could attempt to alleviate this problem using *hybrid* extraction algorithms. Combining our approach with others techniques such as word density or special reasoning could lead to better performance given that these techniques are insensible to the above issues.

Acknowledgments

Acknowledgments to G. Gkotsis from the University of Warwick for generously sharing his research material, time, and ideas with us.

References

1. P. Berger, P. Hennig, J. Bross and C. Meinel, Mapping the Blogosphere — Towards a universal and scalable Blog-Crawler, in *Third Int. Conf. on Social Computing* (2011), pp. 672–677.
2. P. Boldi, B. Codenotti, M. Santini and S. Vigna, UbiCrawler: A scalable fully distributed web crawler, *Software: Practice and Experience* **34**(8) (2003) 711–726.
3. K. Burton, N. Kasch and I. Soboroff, The ICWSM 2011 spinn3r dataset in *Fifth Annual Conference on Weblogs and Social Media* (2011).
4. L. Cantara, Mets: The metadata encoding and transmission standard, *Cataloging & Classification Quarterly* **40**(3-4) (2005) 237–253.
5. D. S. Chung, E. Kim, K. D. Trammell and L. V. Porter, Uses and perceptions of blogs: A report on professional journalists and journalism educators, *Journalism & Mass Communication Educator* **62**(3) (2007) 305–322.
6. L. R. Dice, Measures of the amount of ecologic association between species, *Ecology* **26**(3) (July 1945) 297.
7. S. A. Dudani, The distance-weighted k-nearest-neighbor rule, *IEEE Transactions on Systems, Man and Cybernetics* **SMC-6**(4) (1976) 325–327.
8. N. Eltantawy and J. B. Wiest, Social media in the Egyptian revolution: Reconsidering resource mobilization theory (1-3) (2012).
9. M. Faheem, Intelligent crawling of web applications for web archiving, in *Proc. of the 21st Int. Conf. Companion on World Wide Web* (2012), pp. 127–132.
10. T. Furche, G. Gottlob, G. Grasso, C. Schallhart and A. J. Sellers, Oxpath: A language for scalable data extraction, automation and crawling on the deep web, *VLDB J.* **22**(1) (2013) 47–72.
11. G. Gkotsis, K. Stepanyan, A. I. Cristea and M. Joy, Self-supervised automated wrapper generation for weblog data extraction, in *Proc. of the 29th British National Conference on Big Data* (*BNCOD'13*) (Berlin, Heidelberg, 2013), pp. 292–302.
12. G. Gottlob, C. Koch, R. Baumgartner, M. Herzog and S. Flesca, The lixto data extraction project: Back and forth between theory and practice, in *Proc. of the Twenty-third Symposium on Principles of Database Systems* (2004), pp. 1–12.
13. A. Harasymiv, Blogger dynamic views. `http://buzz.blogger.com/2011/09/dynamic-views-seven-new-ways-to-share.htm`.
14. A. Heydon and M. Najork, Mercator: A scalable, extensible web crawler, *World Wide Web* **2**(4) (1999) 219–229.
15. R. Hundt, Loop recognition in C++/Java/Go/Scala, in *Proc. of Scala Days* (2011).
16. K. Johnson, Are blogs here to stay?: An examination of the longevity and currency of a static list of library and information science weblogs, *Serials Review* **34**(3) (2008) 199–204.
17. H. Kalb and M. Trier, The blogosphere as œuvre: Individual and collective influence on bloggers, in *ECIS 2012 Proceedings*, *Proceedings/European Conference on Information Systems* (*ECIS*) (Association for Information Systems, AIS Electronic Library (AISeL), 2012), Paper 110.
18. N. Kasioumis, V. Banos and H. Kalb, Towards building a blog preservation platform, *World Wide Web* (2013), pp. 1–27.
19. C. Kohlschütter, P. Fankhauser and W. Nejdl, Boilerplate detection using shallow text features, in *Proc. of the Third ACM Int. Conf. on Web Search and Data Mining* (*WSDM '10*) (New York, USA, 2010), pp. 441–450.
20. R. G. Lavender and D. C. Schmidt, *Active Object — An Object Behavioral Pattern for Concurrent Programming* (Addison-Wesley Longman Publishing, Boston, MA, 1996), pp. 483–499.

21. B. Lavoie, Meeting the challenges of digital preservation: The oais reference model, *OCLC Newsletter* **243** (2000) 26–30.
22. V. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics Doklady* **10**(8) (February 1966), 707–710.
23. C. Lindahl and E. Blount, Weblogs: Simplifying web publishing, *Computer* **36**(11) (2003) 114–116.
24. J. G. Llopis *et al.*, D4. 4: Digital repository component design, *work package, European Organization for Nuclear Research* (CERN, 2012).
25. X. MARC, Official web site (2003).
26. I. Muslea, S. Minton and C. A. Knoblock, Hierarchical wrapper induction for semistructured information sources, *Journal of Autonomous Agents and Multi-Agent Systems* **4** (2001) 93–114.
27. M. Oita and P. Senellart, Archiving data objects using web feeds, in *Proc. of the International Workshop on Web Archiving* (*IWAW*) (September 2010).
28. E. Oro, M. Ruffolo and S. Staab, SXPath — Extending XPath towards spatial querying on web documents, *PVLDB* **4**(2) (2010) 129–140.
29. D. C. Reis, P. B. Golgher, A. S. Silva and A. F. Laender, Automatic web news extraction using tree edit distance, in *Proc. of the 13th Int. Conf. on World Wide Web* (*WWW '04*) (New York, USA, 2004), pp. 502–511.
30. A. Rogers and G. Brewer, Microdata usage statistics. `http://trends.builtwith.com/docinfo/Microdata`.
31. RSS Advisory Board. Rss 2.0 specification (2007).
32. V. Shkapenyuk and T. Suel, Design and implementation of a high-performance distributed web crawler, in *18th Int. Conf. on Data Engineering, 2002. Proceedings* (2002), pp. 357–368.
33. Various authors. The reactive manifesto. `http://reactivemanifesto.org`.
34. Various authors, W3C. W3C standards. `http://w3.org/standards`.
35. Various authors, WC3. Use h1 for top level heading. `http://www-mit.w3.org/QA/Tips/Use_h1_for_Title`.
36. WHATWG. Microdata — HTML5 draft standard. `http://whatwg.org/specs/web-apps/current-work/multipage/microdata.html`.
37. B. Wilson, Metadata analysis and mining application. `http://dev.opera.com/articles/view/mama`.
38. WordPress. Posting activity. `http://wordpress.com/stats/posting`.