BlogForever Crawler: Techniques and Algorithms to Harvest Modern Weblogs

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

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

Vangelis Banos
Department of Informatics
Aristotle University of Thessaloniki, Greece
vbanos@gmail.com

ABSTRACT

Blogs are a dynamic communication medium which has been widely established on the web. The BlogForever project has developed an innovative system to harvest, preserve, manage and reuse blog content. This paper presents a key component of the BlogForever platform, the web crawler. 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 and robust algorithm to generate extraction rules based on string matching using the blog's web feed in conjunction with blog hypertext. This approach leads to a scalable blog data extraction process. Furthermore, we show how we integrate a web browser into the web harvesting process in order to support the data extraction from blogs with JavaScript generated content.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering, Query formulation, Selection process*;
D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Performance measures*

General Terms

Design, Algorithms, Performance, Experimentation

Keywords

Blog crawler, web data extraction, wrapper generation

WIMS'14, June 2-4, 2014 Thessaloniki, Greece

1. INTRODUCTION

Blogs disappear every day [13]. Losing data is obviously undesirable, but even more so when this data has historic, political or scientific value. In contrast to books, newspapers or centralized web platforms, there is no standard method or authority to ensure blog archiving and long-term digital preservation. Yet, blogs are an important part of today's web: WordPress reports more than 1 million new posts and 1.5 million new comments each day [32]. Blogs also showed to be an important resource during the 2011 Egyptian revolution by playing an instrumental role in the organization and implementation of protests [6]. The need to preserve this volatile communication medium is nowadays very clear.

Among the challenges in developing a blog archiving software application 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 [19]. 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 [25]. While web feeds essentially solve the question of update monitoring, their limited size makes it necessary to download blog pages in order to harvest previous content.

This paper presents the open-source BlogForever Crawler, a key component of the BlogForever platform [15] responsible for traversing blogs, extracting their content and monitoring their updates. Our main objective in this work is to design a crawler capable of extracting blog articles, authors, publication dates and comments. Our contributions can be summarized as follows:

- We present a new algorithm to build extraction rules from web feeds. We then derive an optimized reformulation tied to a particular string similarity metric and show that this reformulated algorithm has a linear time complexity.
- We show how to use this algorithm for blog article extraction and how it can be adapted to authors, publication dates and comments.
- We present the overall crawler architecture and the specific components we implemented to efficiently traverse blogs. We explain how our design allows for both modularity and scalability.
- We show how we make use of 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.
- We evaluate the content extraction and execution time of our algorithm against three state-of-the-art web article extraction algorithms.

Although our crawler implementation is integrated with the BlogForever platform, the presented techniques and algorithms can be used in other applications related to Wrapper Generation and Web Data Extraction.

2. ALGORITHMS

This section explains in detail the algorithms we developed to extract blog post articles as well as its variations for extracting authors, dates and comments. Our approach uses blog specific characteristics 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.

2.1 Motivation

Extracting metadata and content from HTML documents is a challenging task. Standards and format recommendations have been around for quite some time, strictly specifying how HTML documents should be organised [28]. For instance the `<h1></h1>` tags have to contain the highest-level heading of the page and must not appear more than once per page [29]. More recently, specifications such as microdata [30] 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 [24]. 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 [31]. 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]:

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

Web feeds usually contain about 20 blog posts [21], often less than the total number of posts in blogs. Consequently, in order 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.

2.2 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, to build an extraction rule capable of locating the target content on each HTML page.

Observation (a) 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 the 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 (b) guarantees the existence of an appropriate extraction rule, as well as its applicability to all posts of the blog.

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

[1]Our experiments on a large dataset of blogs showed that failing tests were either due to a violation of one of these observations, or to an insufficient amount of text in posts. *This is for instance the case of photoblogs where posts typically only contain a picture and a few words. Text might then not be sufficient to differentiate the article content from other elements of the page. TODO: The term "insufficient amount of text" needs a couple of lines more explanation. Why does this lead to a failure?*

this section. Subsection 2.3 defines `AllRules`, `Apply` and the `ScoreFunction` we use for article extraction. In subsection 2.4 we analyse the time complexity of Algorithm 1 and give a linear time reformulation using dynamic programming. Finally, subsection 2.5 shows how the `ScoreFunction` can be adapted to extract authors, dates and comments.

2.3 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 Sørensen–Dice coefficient similarity [4], which is, to the best of our knowledge, the only string similarity algorithm fulfilling the following criteria:

- Has low sensitivity to word ordering,
- Has low sensitivity to length variations,
- Runs in linear time.

Properties AllRules and AllRules 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 Sørensen–Dice coefficient similarity but do not for *edit distance* based similarities such as the Levenshtein [18] similarity.

The Sørensen–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:

2.4 Time complexity and linear reformulation

string1	*string2*	Dice	Leven.
`"Scheme Scala"`	`"Scala Scheme"`	90%	50%
`"Rachid"`	`"Richard"`	18%	61%
`"Rachid"`	`"Amy, Rachid and all their friends"`	29%	31%

Table 1: Examples of string similarities

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*

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's 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 $\mathcal{O}(n)$ *TODO: A definition of n is missing (although it can be guessed). I'm for leaving it like this, or changing $\mathcal{O}(n)$ for "a linear number of". What do you think?* 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 bigrams (set of pairs 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 bigrams is not accessible, and the bigrams of the page have to be computed by linearly traversing the entire string. To overcome this limitation and implement *memoization* over the bigrams computations, Algorithm 2 uses a post-order traversal of the HTML tree and computes node bigrams from their children bigrams. This way, we avoid serializing HTML subtrees for each bigrams computation and have the guarantee that each character of the HTML page will be read at most once during the bigrams computation.

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* ⟵
      $\frac{2\,|(bigrams\ \textbf{of}\ node) \cap (bigrams\ \textbf{of}\ target)|}{|bigrams\ \textbf{of}\ node| + |bigrams\ \textbf{of}\ target|}$
  bestRules ⟵ *bestRules* + `Rule(`node with best *score*`)`
return rule with highest occurrence in *bestRules*

With bigrams computed in this dynamic programming manner, the overall time to compute all `Bigrams(`*node.text*`)` is linear. To conclude the proof 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.

2.5 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 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`)`.

Our crawler executes this algorithm on each post with an overflow on its comment feed, thus supporting blogs with multiple commenting engines. The comment content is extracted first, allowing us to narrow down the initial filtering by fixing a target number of comments. *TODO: This paragraph is hard to follow! What is "an overflow on its comment feed"? Do you mean comments content arriving by means of AJAX calls? Please explain. The second sentence of the paragraph is also not clear ("..narrow down ...fixint a target..."). Please rewrite this paragraph. I'm for completely removing this paragraph because I these are very unimportant details about comment extraction. The point on "overflow on comment feed" is a repetition of the" Similarly to blog feeds..." sentence in the paragraph before the bullet points: if the comment feed is not full there is no need to extract comments from the page. The fact that we run this algorithm on each page to support multiple commenting engines is really a detail, I've only found one blog during my tests that used two different engines... Regarding the "comment content is extracted first", this is a small optimization I did to first get the exact number of comments by matching the comment content (which is usually more unique and accurate that doing a matching on author/date), and use this number to then accurately get the other fields. Let me know if you think that this is worth including...*

Regarding time complexity, computing the *tree distance* of each node of a graph to a single reference node can be done in linear time, and multiplying the number of targets by a constant factor does not affect the asymptotic computational complexity. Computing scores of comment extraction rules requires a more expensive algorithm. However, this is compensated by the fact that the proportion of candidates 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` in order to minimize the time spent in `Similarity`.

3. ARCHITECTURE

This section provides an overview of the crawler system architecture and the different techniques we used. The overall

software architecture is presented and discussed, introducing the Scrapy framework and the enrichments we implemented for our specific usage. Then, we show how we integrated a headless web browser into the harvesting process to support blogs that use JavaScript to display page content. Finally, we talk about the design choices we made in view of a large scale deployment.

3.1 Overview

Our crawler is built on top of Scrapy[2], an open-source framework for web crawling. Scrapy provides an elegant and modular architecture illustrated in Figure 1. Several components can be plugged into the Scrapy core infrastructure: *Spiders*, *Item Pipeline*, *Downloader Middlewares* and *Spider Middlewares*; each allowing us to implement a different type of functionality.

Our use case has two types of spiders: *NewCrawl* and *UpdateCrawl*, which implement the logic to respectively crawl a new blog and get updates from a previously crawled blog. After being downloaded and identified as blog posts (details in subsection 3.2), pages are packed into *Items* and sent through the following pipeline of operations:

1. Render JavaScript
2. Extract content
3. Extract comments
4. Download multimedia files
5. Propagate resulting records to the back-end

This pipeline design provides great modularity. For example, disabling JavaScript rendering or plugging in an alternative back-end can be done by editing a single line of code.

*TODO: Is the modularity mentioned available in the current system implementation? For example, can you "sense" somehow the presence of JavaScript content so as to exclude the "Render JavaScript" step in the pipeline? I'm not really sure what to do here. As written before, the modularity *is* available, but at the source code level. I did not do any JavaScript "sensing" as the JavaScript rendering phase also takes care of taking screenshots which we want to do on each page... See the added (*) sentence under 3.3.*

3.2 Enriching Scrapy

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. Internally, for each blog, this function *automatically builds a minimal regular expression that matches all the blog post URLs found in the feed, which is later used to classify URLs. Our implementation does not operates at the granularity of characters, as this sometimes leads to overly precise regular*

[2] `http://scrapy.org/`

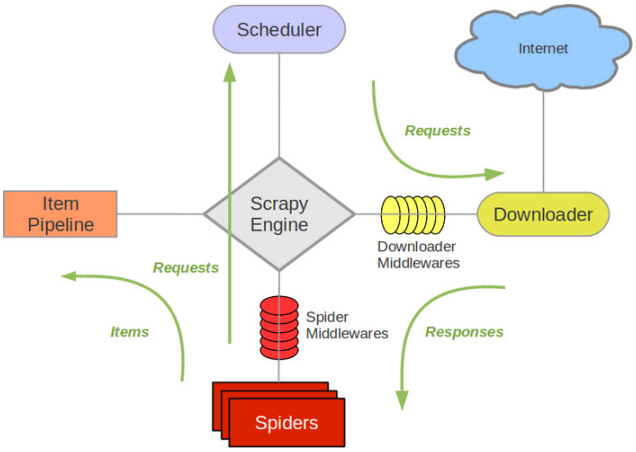


Figure 1: Overview of the crawler architecture. (Credit: Pablo Hoffman, Daniel Graña, Scrapy)

expression which are not be valid for all blog post URLs, such as when the year is part of the URLs. Instead, we restricted the building blocks of these regular expression to sequences of digits, sequences of alphanumeric characters and special characters. 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.

(I also have this reference, but I think it would be overkill here: "While this problem is known to be NP-Hard [22], in practice both the number of feed entries and the length of URLs are bounded.")

TODO: who is building the regular expressions out of the blog post URLs? Is it done manually or automatically? Please clarify. Additionally, what if the blog does not use friendly URLs (which they do support well the idea of constructing regular expressions out of the URLs)? Consider for example a WordPress based blog with URLs of the form http://<domain>/?p=234. How such a non informative URL affects your approach?

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 other pages in order to find 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 useful for blogs hosted on a single domain alongside large number of other types of web pages, such as those of a forum or a wiki. *It also allows the crawler to extract data in presence of *spider traps*, where the naive traversals could have simply missed the actual content.*

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 [5]. 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.

*This priority mechanism allow to stop a blog crawl before all it's pages have been visited while maximizing the proportion of blog posts harvested out of the total number of pages downloaded. While a simple termination condition such an upper bound on the number of pages downloaded is mandatory to avoid infinite loops, it is also possible to add termination heuristics such as *stop if the last 1000 downloaded pages contain less than 1% of blog posts*.*

TODO: The priority queue used, prioritizes the URLs based on a machine learning algorithm trained on the basis of the URL and the number of links it contains. a) Is there any assumption here that the links of the page lead to plog posts? Why do URLs with more links are preferred? b) What is the aim here? To prioritize or to prune? The former makes no sense (at the end you will have to deal with all the pages). The latter, if it holds, is not clear - I was expecting to read about some threshold value under which pages are dropped out of the queue.

*(I don't think there is anything more needed about a), I seams clear here that we are only interested about links to blog posts, and we use the *blog post identification* to identify them. URLs with more links are preferred because the ultimate goal is to to visit blog posts, and URLs that are estimated by the machine learning system to have a big number of links to blog posts are more likely to contain such links.)*

3.3 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[3] and LiveFyre[4] comment hosting services. For webmasters, these tools are very handy because the entire commenting 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 [10].

To support blogs with JavaScript-generated content, we embed a full web browser into the crawler. After considering multiple options, we opted for PhantomJS[5], a headless web browser with great performance and scripting capabilities. The JavaScript rendering is the very first step 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. *(*) PhantomJS also allows to take screenshots of the rendered pages, which is one of the functional requirement of the BlogForever platform [14, FR53].*

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.

3.4 Scalability

When aiming to work with a large amount of input, it is crucial to build every layer of a system with scalability in mind [27]. 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*:

$$NewCrawl : \text{URL} \to \mathcal{P}(\text{RECORD})$$
$$UpdateCrawl : \text{URL} \times \text{DATE} \to \mathcal{P}(\text{RECORD})$$

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.

4. EVALUATION

Our evaluation is articulated in two parts. First, we compare the article extraction procedure presented in section 2 with three open-source projects capable of extracting articles and titles from web pages. The comparison will show that our blog-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

[3] `http://disqus.com/websites`
[4] `http://web.livefyre.com`
[5] `http://phantomjs.org`

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 dedicated core with garbage collection disabled. The Git repository for this paper[6] 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[7].

4.1 Extraction success rates

To evaluate article and title extraction from blog posts we compare our approach to three open source projects: Readability[8], Boilerpipe [16] and Goose[9], 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 blog (observations (a) and (b)). Table 2 shows the extraction success rates for article and title on a test sample of 2300 blog posts from 230 blogs obtained from the Spinn3r dataset [3].

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

Table 2: Extraction success rates

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 2 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 blogs.

4.2 Article extraction running times

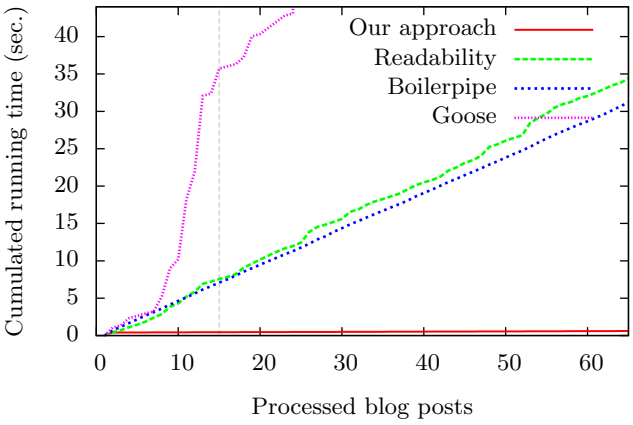
In addition to the quality of the extracted data we also evaluated the running time of the extraction procedure. The main point of interest is the ability of the extraction procedure to scale as the number of posts in the processed blog increases. This corresponds to the evaluation of a *NewCrawl* task,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

[6] `https://github.com/OlivierBlanvillain/bfc-paper`
[7] `https://github.com/BlogForever/crawler`
[8] `https://github.com/gfxmonk/python-readability`
[9] `https://github.com/GravityLabs/goose`

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 function of the number of blog posts processed. We used the Quantum Diaries[10] 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.

Figure 2: Running time of articles extraction.



As illustrated in Figure 2, our approach spends the majority of its total running time between the initialisation and the processing of the first blog 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 of computing the *best extraction rule* of each page referenced by the web feed and picking the most appropriate one. Once we have this extraction rule, processing subsequent blog 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 blog post is processed as new and independent input, leading to approximately linear running times.

The vertical dashed line at 15 processed blog posts represents a suitable point of comparison of processing time per blog post. Indeed, as the web feed of our test blog contains 15 blog posts, the extraction rule computation performed by our approach includes 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 variation of running times observed across programming languages [12].

5. RELATED WORK

Our crawler combines ideas from previous work on general web crawlers and *wrapper* generation algorithms. 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

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

term *extraction rule*, which better reflects our implementation and is decoupled from the XPath engine that concretely performs the extraction.

Web crawling has been a well-studied topic over the past decade. One direction which we believe to be of crucial importance is the one of large scale distributed crawlers. Mercator [11], UbiCrawler [2] and the crawler discussed in [26] are examples of a successful distributed crawler and the papers describing them 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 [11] and [26] 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 circumvents this problem by delegating all shared mutable state to the back-end system. 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 side. This removes one layer of complexity when compared to general crawlers which need to use a distributed file system ([26] uses NFS, [1] uses HDFS) or implement an aggregation mechanism in order to further exploit the collected data. Our design is similar to the distributed active object pattern presented in [17], which is further simplified by the fact that the state of the crawler instances is not kept between crawls.

A common approach in web content extraction is to manually build wrappers for the targeted websites. This approach has been proposed in the crawler discussed in [7] 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 [20] which generates wrappers based on user-labelled training examples. Some commercial solutions such as the Lixto project [9] 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.

Automated solutions use other techniques to identify and extract information directly from the structure and content of the web page. The Boilerpipe project [16] (mentioned in our evaluation) uses text density analysis to extract the main article of a web page. The approach presented in [23] 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 [21] describe a procedure to automatically build wrappers by matching web feed articles with HTML pages. This work was further extended by Gkotsis, Stepanyan, Cristea and Joy [8] with a focus on extracting content anterior to the one indexed in web feeds. [8] also reports 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 [21] nor [8] provide complexity analysis which we believe to be essential before putting an algorithm in production.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented the internals of the BlogForever Crawler. Its central article extraction procedure based on extraction rules generation was introduced along with theoretical and empirical evidence validating the approach. A simple adaptation of this procedure that allows to extract different types of content, including authors, dates and comments was then presented. In order to support rapidly evolving web technologies such as JavaScript-generated content, the crawler uses a web browser to render pages before processing them. We also discussed the overall software architecture, highlighting the design choices made to achieve both modularity and scalability. Finally, we evaluated our content extraction algorithm against three state-of-the-art web article extraction algorithms.

Future work could investigate *hybrid* extraction algorithms to try and achieve near 100% success rates. Indeed, we have observed[11] that the primary causes of failure of our approach were the insufficient quality of web feeds or the high structural variations of blog pages. This suggests that combining our approach with other techniques such as word density or spacial reasoning could lead to better performance given that these techniques are insensible to the above issues.

Another possible research direction would be the deployment of the BlogForever Crawler on a large scale distributed system. 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. We intend to explore these opportunities in our future work.

7. ACKNOWLEDGMENTS

Acknowledgments to our colleagues and friends from CERN, J. Cowton, M. Hobbs and A. Oviedo, for their careful reading and helpful comments that improved the quality of this paper. We are also very grateful to G. Gkotsis from the University of Warwick for generously sharing his research material, time, and ideas with us.

8. REFERENCES

[1] P. Berger, P. Hennig, J. Bross, and C. Meinel. Mapping the Blogosphere–Towards a universal and scalable Blog-Crawler. In *Third International Conference on Social Computing*, pages 672–677, 2011.

[2] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: a scalable fully distributed web crawler. 2003.

[3] K. Burton, N. Kasch, and I. Soboroff. The ICWSM 2011 spinn3r dataset. In *Fifth Annual Conference on Weblogs and Social Media*, 2011.

[11] An in-depth analysis of causes of failure was not included in this paper given the high amount of manual work required to identify causes of failure on problematic pages.

[4] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297, July 1945.

[5] S. A. Dudani. The Distance-Weighted k-Nearest-Neighbor rule. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-6(4):325–327, 1976.

[6] N. Eltantawy and J. B. Wiest. Social media in the egyptian revolution: Reconsidering resource mobilization theory (1-3), 2012.

[7] M. Faheem. Intelligent crawling of web applications for web archiving. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 127–132, 2012.

[8] G. Gkotsis, K. Stepanyan, A. I. Cristea, and M. Joy. Self-supervised automated wrapper generation for weblog data extraction. In *Proceedings of the 29th British National Conference on Big Data*, BNCOD'13, pages 292–302, Berlin, Heidelberg, 2013.

[9] G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The lixto data extraction project: Back and forth between theory and practice. In *Proceedings of the Twenty-third Symposium on Principles of Database Systems*, pages 1–12, 2004.

[10] A. Harasymiv. Blogger dynamic views. `http://buzz.blogger.com/2011/09/dynamic-views-seven-new-ways-to-share.htm`. Last visited 31 Mar 2014.

[11] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *Word Wide Web*, 1999.

[12] R. Hundt. Loop recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days*, 2011.

[13] 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):199–204, 2008.

[14] H. Kalb, N. Kasioumis, J. Garcia Llopis, S. Postaci, S. Arango-Docio, I. Trochidis, and V. Banos. Blogforever: D4.1 user requirements and platform specifications report. Technical report, 2012.

[15] N. Kasioumis, V. Banos, and H. Kalb. Towards building a blog preservation platform. *World Wide Web*, pages 1–27, 2013.

[16] C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate detection using shallow text features. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, page 441–450, New York, NY, USA, 2010.

[17] R. G. Lavender and D. C. Schmidt. Active object – an object behavioral pattern for concurrent programming. 1996.

[18] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710, Feb. 1966.

[19] C. Lindahl and E. Blount. Weblogs: simplifying web publishing. *Computer*, 36(11):114–116, 2003.

[20] I. Muslea, S. Minton, and C. A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.

[21] M. Oita and P. Senellart. Archiving data objects using web feeds. Sept. 2010.

[22] L. Pitt and M. K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, Jan. 1993.

[23] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 502–511, New York, NY, USA, 2004.

[24] A. Rogers and G. Brewer. Microdata usage statistics. `http://trends.builtwith.com/docinfo/Microdata`. Last visited 31 Mar 2014.

[25] RSS Advisory Board. Rss 2.0 specification. 2007.

[26] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *18th International Conference on Data Engineering, 2002. Proceedings*, pages 357–368, 2002.

[27] Various authors. The reactive manifesto. `http://reactivemanifesto.org`. Last visited 31 Mar 2014.

[28] Various authors, W3C. W3C standards. `http://w3.org/standards`. Last visited 31 Mar 2014.

[29] Various authors, WC3. Use h1 for top level heading. `http://www-mit.w3.org/QA/Tips/Use_h1_for_Title`. Last visited 31 Mar 2014.

[30] WHATWG. Microdata - HTML5 draft standard. `http://whatwg.org/specs/web-apps/current-work/multipage/microdata.html`. Last visited 31 Mar 2014.

[31] B. Wilson. Metadata analysis and mining application. `http://dev.opera.com/articles/view/mama`. Last visited 31 Mar 2014.

[32] WordPress. Posting activity. `http://wordpress.com/stats/posting`. Last visited 31 Mar 2014.