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 to blog hypertext. This approach is leading 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.

General Terms

Algorithms, Software Engineering

Keywords

Web crawling, Web Data Extraction, Wrapper Generation, Web archiving, ...

1. INTRODUCTION

Blogs disappear every day. 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 like Facebook, 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 33 million new posts and 48 million new comments each month [31]. 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. [12]. The need to preserve this volatile communication medium is nowadays very clear.

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. 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. Nevertheless, while web feeds essentially solve the question of update monitoring, their limited size makes it necessary to download blog web pages in order to harvest previous content.

1.1 Contributions and Overview

We present the BlogForever Crawler, a key component of the BlogForever platform responsible for traversing blogs, extracting their content and monitoring their updates. Our contributions are in particular:

- 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 process-

ing them. This step allows our crawler to effectively harvest blogs built with modern technology, such as the increasingly popular third-party commenting systems.

- We present and analyze performance results of our algorithm in terms of extraction success rates and execution time. A comparison is made with three state-of-the-art 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. RELATED WORK

The BlogForever Crawler combines ideas from previous work on general web crawlers and extraction rule[1] generation algorithms.

Research on large scale distributed crawler such as Mercator [19], used by AltaVista, the spider discussed in [25] or UbiCrawler [7] are enlightening regarding the difficulties arising when scale out a crawler. One of the core issues seems to be in sharing the list URLs which have already been visited and need to be visited. While [19] and [25] rely on a central node to hold this information, [7] uses a fully distributed architecture where URLs are divided among nodes using consistent hashing. Both of these approaches require complex mechanisms to achieve fault tolerance. The BlogForever crawler circumvent this problem by delegating all state the back-end system. This shared nothing architecture between crawler instance is made possible by the fact that our unit of computations, crawling a entire blog, is much large than fetching a single URL and does not generate further crawling tasks. In addition, because 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 crawler which need use a distributed file system or implement an aggregation mechanism in order to further use the collected data.

[25] used NFS, [6] uses HDFS

- Wrapper generation
these works do not clearly address how to extract other info link author and publication date, and do not include complexity analysis which is something that cannot be ignored when aiming for a universal large scale solution.

[1]In the literature, procedures to extract structured data from unstructured document are commonly referred as *wrappers*. In this paper we decided to rather employ the less generic term of *extraction rules* which is closer to our implementation.

3. ALGORITHMS

This section explains in detail the algorithms we developed to extract blog post articles as well as its variations for authors, dates and comments. Our approach uses blog specific characteristics to build extraction rules which are applicable to all posts of throughout a blog. Our focus is on minimizing the algorithmic complexity while keeping our approach simple and generic.

3.1 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 [27]. 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 [26]. More recently, specifications such as microdata [28] 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 [23]. 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 [29], and more than 95% of pages do not pass HTML validation [30]. 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[2]:

(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 [20], 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 web feed.

3.2 Content extraction

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 builder 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.

[2]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.

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 HTML 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 programing. 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 favor `class` over path, and `id` over `class`, such that the `AllRules` function returns a single rule per node.

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 [9], 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.

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*

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 Sørensen–Dice coefficient similarity but do not for *edit distance* based similarities such as the Levenshtein [17] similarity.

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

The Sørensen–Dice coefficient similarity algorithm operates by first building sets of pairs of adjacent characters, also knows 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

The functions `AllRules`, `Apply` and `Similarity` (as `ScoreFunction`) being defined, the definition of Algorithm 1 for article extraction is complet. We now 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 $\mathcal{O}(n)$ 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. 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.

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 previously presented `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 inconsistency 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 on `"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 the post web page. To do so we use the following `ScoreFunction`:

- Rules returning less 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.

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`.

4. TECHNIQUES

This section gives an overview of the different techniques used in our crawler. We first show how we integrated a headless web browser into the harvesting process to support blogs that use JavaScript to display page content. The overall software architecture is then discussed, introducing the Scrapy framework and the enrichments we implemented for our specific usage. Finally we talk about the design choices we made in view of a large scale deployment.

4.1 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 [10] and LiveFyre [18] 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 [14].

To support blogs with JavaScript-generated content, we embed a full web browser into the crawler. After considering multiple options, we opted for PhantomJS [21], 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.

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.

4.2 Architecture

Our crawler is built on top of Scrapy [24], an open-source Python framework for web crawling. Scrapy provides an elegant and modular architecture illustrated in Figure 1. Several components can be plugged into the Scrapy core

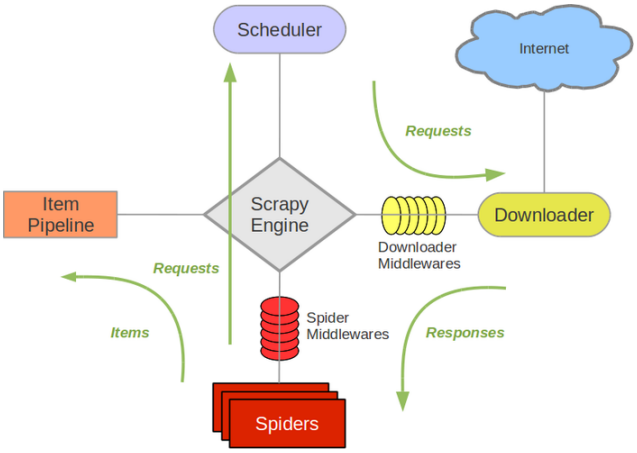


Figure 1: Overview of the crawler architecture. (Credit: Pablo Hoffman, Daniel Graña, [24])

infrastructure: *Spiders*, *Item Pipelines*, *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 4.3), 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 one single line of code.

TODO: Write this properly... One of the main advantages of having full control over the crawling process is that contrarily to the general purpose crawlers, we do not to store all the web pages material. Instead, the content of interest is computed "on the fly", and directly saved into the back-end system. This propriety ...

4.3 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 an URL entry point to a website, the default Scrapy behavior 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 a 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 amongst 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-Neighbor* classifier [11]. 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 a 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 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 [4]. 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*:

$$NewCrawl : \mathbb{U}\text{RL} \to \mathcal{P}(\mathbb{R}\text{ECORD})$$
$$UpdateCrawl : \mathbb{U}\text{RL} \times \mathbb{D}\text{ATE} \to \mathcal{P}(\mathbb{R}\text{ECORD})$$

where $\mathbb{U}$RL, $\mathbb{D}$ATE and $\mathbb{R}$ECORD are respectively the set of all URLs, dates and records, and $\mathcal{P}$ designates the powerset operator. By delegating all mutable state to the back end system, web crawler instances can be added, removed and used interchangeably.

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 article and title 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 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 [5] 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 [2].

5.1 Extraction success rates

To evaluate article and title extraction from blog posts we compared our approach to three open source projects: Readability [22], Boilerpipe [16] and Goose [13], 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 (1) and (2)). Table 2 shows the extraction success rates for article and title on a test sample of 2300 blog posts obtained from the Spinn3r dataset [8].

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 showen in Table 2 should not be interpreted as a general quality evaluation of the different projects, but simply as an evidence that our approach is more suitable when working with blogs.

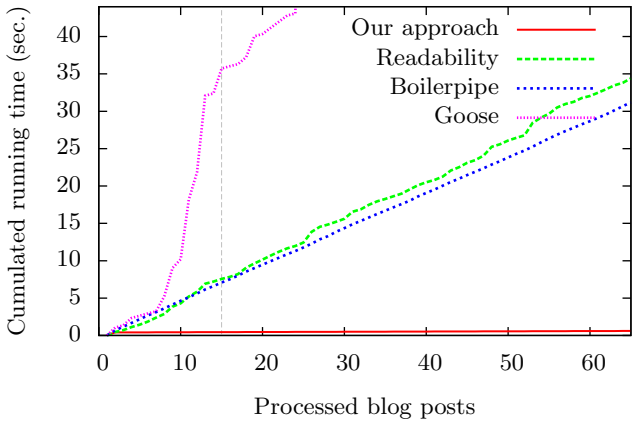
5.2 Article extraction running times

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 blog increases. This corresponds to the evaluation of a *NewCrawl* task, which is in charge of harvesting all published content on a blog.

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 [3] 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



Past the initial computations, the cost of processing a blog post with our approach is almost zero. This is a consequence of having a blog-aware, rule-based algorithm. As already mentioned, the central idea is to first build extraction rules using the information provided by the web feed, and then use these rules on all posts of the blog. The initial increase in the curve of our approach corresponds to the computation of the extraction rules, which consists of processing the web feed and all the blog posts it references. Subsequent computations only involve parsing blog posts and applying extraction rules, 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 as the test blog's web feed contains 15 blog posts. That being said, comparing raw performance of different algorithms implemented in different programming languages is not very informative given the high variations of running times observed across languages for identical algorithms [15].

5.3 Preservation of blog visuals

TODO: visuals/design/graphic/?

Additionally to the blog content extraction that we have been discussing so far, the BlogForever crawler is also responsible for capturing blog visuals. Indeed, one of the requirements

of the BlogForever platform is to provide snapshots of blogs visuals in order to preserve changes in appearance [1, FR53]. Our final solution consists in using the embeded web browser to take screenshots after each page rendering. Before opting for this approach we considered 3 other solutions: GNU wget, wkhtmltopdf and OXPath. This subsection relates our evaluation of these tools for the specific purpose.

TODO: Is this whole subsection legitimate? It this part is not really relevant we could also simply not include it... I guess it is not a really useful from a research point of vue, but it might still be of some use for others to know what each of them can do. I think it's also important to say at some point how long it takes to render a pages, because if only look at the graph it seams like this is all super fast, but when we use PhantomJS the thing seriously slows down...

The first solution we considered was the GNU/Linux tool wget. With the appropriate options, wget can be used to download a webpage and the associated files (css, scripts and images). On simple webpages, this process effectively captures all files necessary to display the page locally. However, it falls short* to capture webpages that download content via javascript.

Another solution we considered is wkhtmltopdf, a tool to produce pdf versions of web-pages. It uses the "print" function of it's embedded web browser to produce it's output and therefore supports all web technologies supported by it underling web browser.

The OXPath project goes one step further by offering an extension of the XPath language to specify page interactions such as clicks and forms filling. It also make use of a complete web browser to render pages and execute page interactions sequentially. OXPath makes it simple to extract comments displayed across multiple pages or that requiring clicks to be loaded.

Functionality	wget	wkhtml.	OXPath	PhantomJS
Page Interaction	✗	✗	✓	✓
JavaScript	✗	✓	✓	✓
Running time	0.50s	7.58s	22.04s	3.32s

Table 3: *TODO: some text*

6. CONCLUSION AND FUTURE WORK

In this paper, we presented the internals of the BlogForever web 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. *TODO: Finally, ... evaluation*

Future work could investigate *hybrid* extraction algorithms to try and achieve near 100% success rates. Even if the overall performance of our approach is better than those of comparable projects (as shown in the evaluation), there are still a few cases where other techniques managed to extract data and we did not. This suggests that combining our approach with others such such as word density, tree edit distance matching or even spacial reasoning could lead to better performance.

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. Crawler greatly benefits from the use of multiple Internet access points which makes them natural candidates for distributed computing. We intend to explore this opportunitie in our future work.

7. REFERENCES

[1] Blogforever: User requirements and platform specifications report. http://goo.gl/Rjn5wT, 2009.

[2] Blogforever crawler project page. https://github.com/BlogForever/crawler, 2014.

[3] Quantum diaries. http://www.quantumdiaries.org/, 2014.

[4] The reactive manifesto. http://reactivemanifesto.org/, 2014.

[5] Repository of this paper. http://goo.gl/c9N2wX, 2014.

[6] P. Berger, P. Hennig, J. Bross, and C. Meinel. Mapping the Blogosphere–Towards a universal and scalable Blog-Crawler. In *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*, pages 672–677, 2011.

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

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

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

[10] Disqus. http://disqus.com/websites/, 2014.

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

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

[13] Goose. https://github.com/GravityLabs/goose, 2012.

[14] A. Harasymiv. Blogger dynamic views. http://goo.gl/Ma3G2I, 2011.

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

[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. ACM.

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

[18] Livefyre. http://web.livefyre.com/, 2014.

[19] M. Najork, A. Heydon, M. Najork, and A. Heydon. High-performance web crawling. Technical report, SRC Research Report 173, Compaq Systems Research, 2001.

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

[21] PhantomJS. http://phantomjs.org/, 2014.

[22] python readability. https://github.com/gfxmonk/python-readability, 2011.

[23] A. Rogers and G. Brewer. Microdata usage statistics. http://trends.builtwith.com/docinfo/Microdata, 2014.

[24] Scrapy. http://scrapy.org/, 2014.

[25] 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.

[26] W3C. Use <h1> for top level heading. http://www-mit.w3.org/QA/Tips/Use_h1_for_Title, 2002.

[27] W3C. W3C standards. http://www.w3.org/standards/, 2014.

[28] WHATWG. Microdata - HTML5 draft standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html, 2014.

[29] B. Wilson. Metadata analysis and mining application: Common attributes. http://dev.opera.com/articles/view/mama-common-attributes/, 2008.

[30] B. Wilson. Metadata analysis and mining application: W3C validator research. http://dev.opera.com/articles/view/mama-w3c-validator-research-2/, 2008.

[31] WordPress. Stats, 2014.