Scraping Data from Web Pages using SPARQL Queries

Radek Burget[1][0000−0001−5233−0456]

Brno University of Technology, Faculty of Information Technology, Bozetechova 2, 61266 Brno, Czechia
burgetr@fit.vut.cz

Abstract. Despite the increasing use of semantic data, plain old HTML web pages often provide a unique interface for accessing data from many domains. To use this data in computer applications or to integrate it with other data sources, it must be extracted from the HTML code. Currently, this is typically done by single-purpose programs called scrapers. For each data source, specific scrapers must be created, which requires a thorough analysis of the source page's implementation in HTML. This makes writing and maintaining a set of scrapers a complex and time-consuming task. In this paper, we present an alternative approach that allows defining scrapers based on visual properties of the presented content instead of the HTML code structure. First, we render the source page and create an RDF graph that describes the visual properties of every piece of the displayed content. Next, we use SPARQL to query the model and extract the data. As we demonstrate with real-world examples, this approach allows us to easily define more robust scrapers that can be used across multiple web sites and that that better cope with changes in the source documents.

Keywords: Web scraping · Page rendering · Data extraction · RDF · SPARQL.

1 Introduction

Web pages often provide a unique interface for accessing data from many domains, such as e-commerce, news, movies, real estate, and a vast number of others. Data presented in this way is easily accessible to human readers, but due to the semi-structured nature of the HTML language used, integrating web data sources with computer applications or other data sets is a challenging task.

In recent years, we have seen an increase in the use of structured data annotations created using JSON-LD, Microdata, or RDFa along with semantic vocabularies such as schema.org. According to Web Data Commons statistics, structured data was available in 42 % of the 33.8 million domains included in the Common Crawl corpus [3]. However, the use of structured data is growing slowly and is limited to certain domains. A large part of the annotated documents come from the major content providers such as Google, while there remains a large set

of diverse "long tail" data sources whose providers are less motivated to provide the structured data.

Therefore, it is a common practice in web data integration today to use special programs called *scrappers* that extract the required data directly from the HTML code of the source pages. Currently, scrappers are typically tailored for a specific web site and identify the required data fields based on rules that have been designed by an analyst based on an examination of the HTML code. Creating scrapers requires considerable website implementation expertise, they are limited to very narrow sets of web pages, and they are very sensitive to changes in the source web pages, which can occur at any time.

In this paper, we present an alternative approach that allows defining scrapers based on visual properties of the presented content instead of the HTML code structure. Key advantages of this approach are (1) There is no need to analyze the details of the HTML code as the extraction rules are specified by the visual and textual properties of the content, and (2) the rules can be made more general, making scrapers more resistant to anomalies and changes in the source documents, and even making it possible to define scrapers that can be applied across multiple web sites.

The proposed solution is built on top of existing, widely available technologies. We use the Resource Description Framework (RDF) to represent all the details of the source page content, including its visual representation, and later SPARQL is used to query the models and extract the data. Due to the growing popularity of the semantic web, many developers are already familiar with these technologies, which we consider another advantage of the presented solution.

2 Related Work

The extraction of data from HTML documents has been the subject of research for more than 25 years. Despite this long history of research, ranging from the early string-based *wrappers* [1], through the most common DOM-oriented methods [2, 6], to the sophisticated applications of machine learning methods [9, 11], it is still a widely accepted practice in the industry to create procedural scrapers to perform this task. By a *scraper* we understand a single-purpose program (or procedure) that takes HTML code as its input and produces the extracted structured values as its output. Typically, scrapers are written in general-purpose languages (most often Python, JavaScript, and Ruby are mentioned in this context [4]), often combined with the use of CSS selectors or XPath to select important DOM nodes from the source page model.

The vulnerability to errors caused by changes and variations in the source documents has been a well-known feature of traditional scrapers for a long time [10]. Several approaches have been proposed that aim to provide a more robust solution. For example, in [13], the initial DOM node selection phase (based on node classification) is followed by an additional *visual validation* phase to eliminate incorrectly extracted nodes. In [7], robustness is improved by not strictly

applying the XPath expressions, but by evaluating the similarity of the potential paths.

With the development of machine learning algorithms, their application to web data extraction has received increasing attention. A number of methods have been developed that use different neural network architectures for this task [8, 9, 11]. The results show that such approach is usable for extracting data from large sets of diverse web sites. On the other hand, such extractors are quite complex and require the preparation of large annotated data sets for training the neural networks.

Although Semantic Web technologies such as RDF and SPARQL are closely related to web development and are often used for the definition of structured data, their use for the scraping of web content has been very rare. In [5], the authors propose an RDF-based framework for mapping web content fragments to RDF resources, and the *visual selectors* can be used in addition to XPath and CSS selectors to address DOM nodes. However, the visual properties considered include only the basic font and color properties of the node itself, which does not allow, for example, taking advantage of the mutual positions of different elements on the page. In [12], we proposed an ontological model that allows to capture different aspects of web page content at different levels of abstraction, and we used it to extract domain-specific content from the web using a set of ad hoc heuristic rules. In this paper, we generalize this approach by using general SPARQL queries to define scrapers for an arbitrary application.

As we will show in the next sections, the proposed approach allows to build web scrapers in a straightforward way, without the need to manually examine the source HTML code. The use of a generic query language allows to abstract from implementation details and to create robust scrapers applicable to variable source pages, while avoiding the complexity and other drawbacks of advanced machine learning methods.

3 Method Overview

Our method operates on a page rendered by a standard web browser (we use Chromium in our implementation, as we describe in section 5). When rendering a page, the browser generates a tree of *boxes*, where a box represents a rectangular area in the rendered page with some content. The browsers generate the boxes from DOM elements in a manner defined by the CSS Visual Formatting Model[1].

The page processing workflow is shown in Figure 1. It assumes a central RDF repository that stores the complete information about the page being processed and its content and on which SPARQL queries are subsequently executed. The RDF representation of this information makes use of the Box Model Ontology[2] and the Visual Area Ontology[3] that we published in [12]. The former defines (among other things) the concept of a *Box*, which corresponds to a box generated

[1] https://www.w3.org/TR/CSS22/visuren.html
[2] http://fitlayout.github.io/ontology/render.owl#
[3] http://fitlayout.github.io/ontology/segmentation.owl#

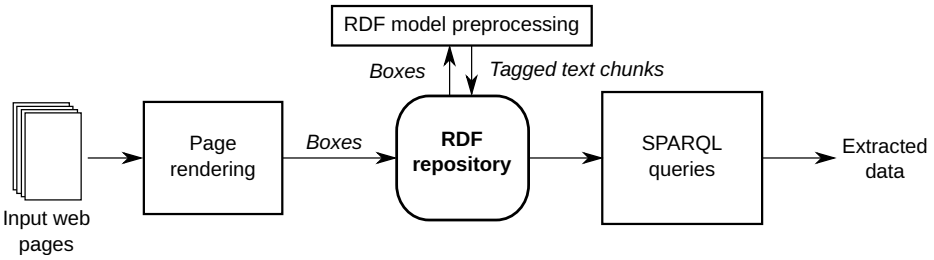

Fig. 1. Overall workflow of data extraction from source web pages using SPARQL queries.

by a web browser during page rendering, and its properties. Similarly, the latter ontology defines the concept of a *Text Chunk*, which represents a part of a box text that has some interpretation (e.g. a product price), with the same properties as the box itself. The relevant properties of boxes and text chunks are summarized in Table 1.

Table 1. Properties of *boxes* and *text chunk* used in their RDF description. The `bounds` property assigns an object with the `positionX`, `positionY`, `width`, and `height` properties to a box or text chunk. The values of these properties are given in pixels with the origin of the coordinate system at the top left corner of the rendered page.

Property name	**Description**
`backgroundColor`	Background color in hex notation
`color`	Text color in hex notation
`fontFamily`	Font family name
`fontSize`	Font size in *pt* units
`fontStyle`	Average font style (0.0 for normal font, 1.0 for italic)
`fontWeight`	Average font weight (0.0 for normal font, 1.0 for bold)
`lineThrough`	Line-through text decoration
`underline`	Underline text decoration
`bounds`	Box or visual area position and size within the page.
`text`	Contained text
`contentLength`	Contained text length in characters
`containsObject`	Description of a contained object such as image

After the page is rendered, we describe the generated boxes (including the values of all their properties) by RDF statements using the Box Model Ontology, and we store the statements in the RDF repository. Next, we preprocess the RDF model. This consists of the following tasks:

Tagged text chunk extraction We select all the boxes that have the `text` property set, create the text chunk instances with the same properties as the original boxes, and add their descriptions to the RDF repository. Optionally, we can

specify a regular expression for each extracted data field that specifies its expected text format (for example, for product price). This allows us to pre-select the boxes that potentially correspond to the expected data, and possibly select a relevant substring of the box text for creating the text chunk. Then, the created chunks are assigned a *tag* that indicates the data field to which they potentially correspond. Note that this is only a rough pre-selection of the boxes and the regular expressions can be very general in this phase; the final extraction is performed later in the query phase.

Discovery of spatial relationships among areas We go through the list of created text chunks and we look for the pairs of text chunks that are located below each other while their x-coordinates overlap at least partially. For each such pair $(c1, c2)$ we add new statements `c1 below c2` and `c2 above c1` to our RDF model. Similarly, we add statements with the predicates `before` and `after`, which represent a similar relationship in the x-axis direction. For convenience, we also define the equivalent functions `isAbove()`, `isBelow()`, `isBefore()`, and `isAfter()` that can be used in SPARQL predicates.

4 Querying the RDF Model

After preprocessing, the RDF repository contains a complete RDF graph that can be queried using SPARQL. In the queries, both the visual properties of the boxes (as listed in Table 1) and their spatial relationships can be used for identifying the text chunks that contain the required data. The query may include multiple boxes whose properties can be compared arbitrarily.

We demonstrate the use of SPARQL for this task on two typical scenarios: extracting multiple records from a single input page and extracting single records from a large set of pages from different web sites.

4.1 Extraction of Multiple Records

To demonstrate the extraction of multiple repeating records from a single website, we use the cast tables on IMDb (Figure 2). For preprocessing, we defined taggers (regular expressions) that roughly recognize the names and the episode strings and assign the tags *name* and *credit* to the corresponding text chunks. We then extract the records using the query shown in Figure 3.

In the query, we first identify the header text chunk (`hbox`) that contains the "Series Cast" text. Below this header, we look for triplets of boxes that match the actor name, character name, and episodes and are arranged on one line after each other.

We chose this particular table for the demonstration because its HTML code is very complex and creating a DOM-based scraper for it is a non-trivial task. On the other hand, the visual presentation is straightforward, which makes the SPARQL code quite simple.

Fig. 2. Top of the source table on IMDb

```
SELECT ?name ?character ?episodes WHERE {
  ?hbox rdf:type segm:TextChunk .
  ?hbox segm:text ?header .
  # Actor names
  ?nbox segm:hasTag r:tag-generic--name .
  ?nbox segm:text ?name .
  # Character names after the header
  ?cbox r:rel-after ?nbox .
  ?cbox segm:hasTag r:tag-generic--name .
  ?cbox segm:text ?character .
  # Episodes after the character
  ?ebox r:rel-after ?cbox .
  ?ebox segm:hasTag r:tag-generic--credit .
  ?ebox segm:text ?episodes .
  # Names are below the header
  FILTER (regex(?header, 'Series␣Cast␣')
    && flfn:isBelow(?nbox, ?hbox))
}
```

Fig. 3. SPARQL query to scrape series cast tables from IMDb

```
SELECT ?tbox ?pbox ?tfsize ?pfsize ?ty ?py WHERE {
    FILTER (?ty < ?py)
    { SELECT ?tbox ?tfsize ?ty ?tw ?th ?ttag WHERE {
            ?tbox segm:hasTag r:tag-generic--title .
            ?tbox box:fontSize ?tfsize . ?tbox box:contentLength ?tlen .
            ?tbox box:bounds ?tb . ?tb box:positionY ?ty . ?tb box:width ?tw . ?tb box:height ?th
            FILTER (?ty <= 500 && ?tlen > 10)
        } ORDER BY DESC (?tfsize) LIMIT 10 }
    { SELECT ?pbox ?pfsize ?py ?pw ?ph ?ptag WHERE {
            ?pbox segm:hasTag r:tag-generic--price .
            ?pbox box:fontSize ?pfsize . ?pbox box:bounds ?pb .
            ?pb box:positionY ?py . ?pb box:width ?pw . ?pb box:height ?ph
            FILTER (?py <= 1200)
        } ORDER BY DESC (?pfsize) LIMIT 10 }
} ORDER BY DESC (?tfsize) DESC (?pfsize) DESC(?th) DESC(?ph) DESC(?tw) DESC(?pw) ?ty ?py LIMIT 1
```

Fig. 4. SPARQL query to extract product title and price from e-commerce product pages

4.2 Extraction from Diverse Web Sites

In this scenario, the task is to extract single (*title*, *price*) pairs from e-commerce web pages. Again, we have prepared taggers that assign the corresponding tags to the extracted pieces of text. The corresponding query in Figure 4 is based on the simple observation that the product title is typically placed at the top of the page (at most 500 px from the top) and the price is always lower than the title (we allow up to 1200 px from the top). Both are written in a larger font than the rest of the content. We look for instances of two different text chunks (`?tbox` for product title and `?pbox` for price). We use a separate subquery for each of these boxes, each of which retrieves up to 10 candidate boxes that match the given condition, ordered by font size, starting with the largest font size.

In the main query, we combine the results of both subqueries, considering only the combinations where the title box is above the price box (`?ty` $<$ `?py`). The final order ensures that we favor the boxes with the largest font sizes, the largest width and height, and the smallest Y-positions. We use the first result in the resulting list.

5 Implementation and Experimental Results

To test the queries in a practical setting, we have implemented an experimental application[4] that allows to process any input page and execute the given query on the created RDF model. The implementation is based on our generic FitLayout framework[5], which uses the Playwright library[6] and Chromium for rendering the pages, creates the RDF models, and uses RDF4J[7] as the RDF repository.

We tested the Cast table extraction on 100 IMDb tables, from which we correctly extracted 43624 records. Another 2060 records were omitted because their visual appearance did not match the SPARQL query (overlapping columns in the table). This problem can be solved at the cost of making the query more complex. The product/price extraction was tested on a total of 1898 product pages from 5 different e-commerce sites that use different visual presentation of products (and very different HTML code). Both title and price were correctly extracted in all cases. The extraction results for both scenarios, including annotated screenshots of the pages, are available in a separate git repository[8].

The visual presentation-based approach using a full-featured web browser in the background implies an increased complexity of the solution. Rendering and processing a single page took from 9 seconds[9] for the e-commerce pages and short Cast tables up to 6 minutes for an extremely long page[10]. However, we believe that the time complexity can be reduced by a more precise selection of the target area of the page, as we have demonstrated for e-commerce sites, where we strictly limit the analyzed area. The browser-based solutions such as Playwright or Selenium are already being used for web scraping for a variety of other reasons, making it easy to integrate the SPARQL approach.

6 Conclusions

In this paper, we have presented an approach to define web scrapers using SPARQL queries over an RDF model of the rendered page. As we show through our experimental implementation and results, scrapers defined in this way are general enough to be applied in different scenarios including a set of different web sources with similar visual presentation. This also implies the robustness of the scrapers with respect to changes and deviations in the source documents. In addition, no knowledge of the HTML code is required to create the scrapers. We consider these features to be an advantage over traditional scrapers, which are tailored to a specific page code. The implemented solution is based on the technology already used in this area, which allows its integration into the existing infrastructure.

[4] https://github.com/FitLayout/sparql-web-scraping
[5] https://github.com/FitLayout/FitLayout
[6] https://playwright.dev/
[7] https://rdf4j.org/
[8] https://github.com/FitLayout/sparql-web-scraping-results
[9] On Intel(R) Core(TM) i5-9500 CPU 3.00GHz, 16 GB RAM
[10] The rendered page height was 183,294 pixels.

Acknowledgements The work is supported by the Brno University of Technology project "Application of AI methods to cyber security and control systems", no. FIT-S-20-6293.

References

1. Ashish, N., Knoblock, C.A.: Wrapper generation for semi-structured internet sources. SIGMOD Rec. **26**(4), 8–15 (Dec 1997). https://doi.org/10.1145/271074.271078
2. Baumgartner, R., Flesca, S., Gottlob, G.: Visual web information extraction with Lixto. In: VLDB '01. pp. 119–128. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
3. Bizer, C., Meusel, R., Primpeli, A., Brinkmann, A.: Web data commons - microdata, RDFa, JSON-LD, and microformat data sets - extraction results from the october 2022 common crawl corpus (2022), http://webdatacommons.org/structureddata/2022-12/stats/stats.html, accessed on 2023-01-29
4. Dilmegani, C.: Best web scraping programming languages in 2023 with stats (2023), https://research.aimultiple.com/web-scraping-programming-languages/, accessed on 2023-02-05
5. Fernández-Villamor, J.I., Blasco-García, J., Iglesias, C.A., Garijo, M.: A semantic scraping model for web resources - applying linked data to web page screen scraping. In: Proceedings of ICAART 2011. Roma, Italia (2011)
6. Furche, T., Gottlob, G., Grasso, G., Schallhart, C., Sellers, A.: OXPath: A language for scalable data extraction, automation, and crawling on the deep web. The VLDB Journal **22**(1), 47–72 (Feb 2013). https://doi.org/10.1007/s00778-012-0286-6
7. Gao, P., Han, H.: Robust web data extraction based on weighted path-layer similarity. Journal of Computer Information Systems **62**(3), 536–546 (2022). https://doi.org/10.1080/08874417.2020.1861571
8. Gogar, T., Hubacek, O., Sedivy, J.: Deep Neural Networks for Web Page Information Extraction. In: 12th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI). vol. AICT-475, pp. 154–163. Thessaloniki, Greece (Sep 2016). https://doi.org/10.1007/978-3-319-44944-9_14
9. Hotti, A., Risuleo, R.S., Magureanu, S., Moradi, A., Lagergren, J.: Graph neural networks for nomination and representation learning of web elements (2021). https://doi.org/10.48550/ARXIV.2111.02168
10. Kushmerick, N.: Wrapper verification. World Wide Web **3**(2), 79–94 (2000). https://doi.org/10.1023/A:1019229612909
11. Lin, B.Y., Sheng, Y., Vo, N., Tata, S.: FreeDOM: A transferable neural architecture for structured information extraction on web documents. In: KDD '20. p. 1092–1102. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3394486.3403153
12. Milička, M., Burget, R.: Information extraction from web sources based on multi-aspect content analysis. In: Semantic Web Evaluation Challenges, SemWebEval 2015 at ESWC 2015. pp. 81–92. No. 548 in Communications in Computer and Information Science, Springer International Publishing, Portorož, SI (2015)
13. Potvin, B., Villemaire, R.: Robust web data extraction based on unsupervised visual validation. In: Intelligent Information and Database Systems. pp. 77–89. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-14799-0_7