- WORKING DRAFT -

Squirrel – Crawling RDF Knowledge Graphs on the Web

Michael Röder[1,2][0000−0002−8609−8277], Geraldo de Souza[1], and Axel-Cyrille Ngonga Ngomo[1,2][0000−0001−7112−3516]

[1] Department of Computer Science, Paderborn University, Germany
michael.roeder|axel.ngonga@upb.de
[2] Institute for Applied Informatics, Leipzig, Germany

Abstract. The growth of the Semantic Web leads to an increased need for crawlers to gather RDF knowledge graphs on the Web. However, existing open-source Linked Data crawlers have limitations with respect to the type of data they are able to crawl. We hence present Squirrel, an open-source distributed crawler for the Semantic Web which supports a wide range of RDF serializations as well as additional structured and semi-structured data formats. Squirrel has been used in the context of several research projects and is freely available at https://github.com/dice-group/squirrel.

1 Introduction

The Semantic Web grew during the last years to comprise several thousands of RDF datasets [3].[3] Additionally, the value of data and the number of applications for linked data has grown over the last years. Hence, in several use cases users could benefit from this freely available data. Since manually searching and analysing these datasets is not feasible, data web crawlers are needed. However, there is only a limited number of freely available open-source crawlers that could be used for this task and the available crawler have crucial limitations. We close this gap by presenting Squirrel—an open-source scalable distributed crawler for the web of data.[4] Squirrel supports a wide range of RDF serializations, decompression algorithms and formats of structured data. The crawler is designed to use Docker[5] containers providing a simple build and run architecture [12]. Squirrel is built using a modular architecture and the concept of dependency injection using Spring Beans.[6] This allows for a further extension of the crawler and an adaption to different use cases.

[3] See https://lod-cloud.net/ for an example of a growing network of RDF datasets.
[4] The code is available at https://github.com/dice-group/squirrel and the documentation at https://w3id.org/dice-research/squirrel/documentation.
[5] https://www.docker.com/
[6] https://spring.io/projects/spring-framework

Table 1. A comparison of RDF serialisations, formats of structured data embedded in HTML and compression formats supported by LDSpider [9] and Squirrel.

	RDF Serialisations										**Emb.**				**Comp.**			
	RDF/XML	RDF/JSON	Turtle	N-Triples	N-Quads	Notation 3	JSON-LD	TriG	TriX	HDT	RDFa	Microdata	Microformat	JSON-LD	ZIP	Gzip	bzip2	tar
LDSpider	✓	–	✓	✓	✓	✓	✓	–	–	–	✓	✓	✓	✓	–	–	–	–
Squirrel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

The reminder of this paper is structured as follows. We describe related work in Section 2 and the proposed crawler in Section 3. Section 4 shows the applications of Squirrel. We point out future directions and conclude the paper in Section 6.

2 Related work

There is only a small number of open-source Data Web crawlers available which can be used to crawl RDF datasets. Related open-source projects are either not able to process RDF or have limited functionalities compared to Squirrel.

An open-source Linked Data crawler used in several publications to crawl data from the web is LDSpider [9].[7] Table 1 shows the serializations and compressions LDSpider supports. It can make use of several threads in parallel to improve the crawling speed. It offers two crawling strategies. The breadth-first strategy follows a classical breadth-first search approach for which the maximum distance to the seed URI(s) can be defined as termination criteria. The load-balancing strategy tries to crawl URIs in parallel without overloading the servers hosting the data. Hence, the latter strategy is similar to the strategy implemented by Squirrel's frontier implementation. The crawled data can be stored either in files or can be sent to a SPARQL endpoint. In comparison to Squirrel, several RDF serialisations and compression formats are not supported. Apart from that, it can not be deployed in a distributed environment. Another limitation of LDSpider is the missing functionality to crawl SPARQL endpoints and open data portals.

A crawler focusing on structured data is presented in [5]. It comprises a 5-step pipeline and converts structured data formats like XHTML or RSS into RDF. The evaluation is based on experiments in which the authors crawl 100k randomly selected URIs. To the best of our knowledge, the crawler is not available as open source project. In [7,8], a distributed crawler is described, which is used to index resources for the Semantic Web Search Engine. In the evaluation,

[7] https://github.com/ldspider/ldspider

different configurations of the crawler—different numbers of threads as well as machines on which the crawler has been deployed—are compared, based on the time the crawler needs to crawl a given amount of seed URIs. To the best of our knowledge, the crawler is not available as open-source project. In [2], the authors present the LOD Laundromat—an approach to download, parse, clean, analyse and republish RDF datasets. The tool relies on a given list of seed URLs and comes with a robust parsing algorithm for various RDF serialisations. In [3], the authors use the LOD Laundromat to provide a dump file comprising 650K datasets and more than 28 billion triples.

Apache Nutch is an open-source web crawler.[8] However, the only available plugin for processing RDF stems from 2007, relies on an out-dated crawler version and was not working during our evaluation.[9]

The Mercator Web Crawler [6] is an example of a web crawler. The authors describe the major components of a scalable web crawler and discuss design alternatives. The evaluation of the crawler comprises an 8-day run, which has been compared to similar runs of the Google and Internet Archive crawlers. As performance metrics, the number of HTTP requests performed in a certain time period, and the download rate (in both documents per second and bytes per second) are used. Additionally, further analysis is undertaken regarding the received HTTP status codes, different content types of the downloaded data, and which parts of the crawler the most CPU cycles are spent. This publication can be seen as an example of a classical crawler evaluation, which comes with the drawbacks explained in the previous Section.

3 Approach

In order to deliver a robust, distributed, scalable and extensible data web crawler we pursue the following goals with Squirrel:

- The crawler should be designed to provide a distributed and scalable solution on crawling structured and semi-structured data. This is achieved by designing Squirrel as a distributed crawler which allows an easy horizontal scaling.
- Squirrel needs to have a gently behavior when fetching data from servers, by following the Robots Exclusion Standard Protocol [10], to make sure that the server is not overloaded and the crawler is not blocked by the server, by not disobeying the rules from robots.txt.
- Although Squirrel is a Linked Data crawler, a requirement for the crawler was to be able to support the processing of webpages that do not contain any RDF data by supporting scraping. Hereby, scraping is defined as the extraction of structured data from a webpage based on rules which use the HTML structure of a webpage to derive the meaning of its content.

[8] http://nutch.apache.org/
[9] https://issues.apache.org/jira/browse/NUTCH-460

– The project should offer easy addition of further functionalities, with a fully extensible architecture.
– The crawler should provide metadata about the cralwing process, allowing the users to get insights from the crawled data later.

The next section will go into details about the crawler components and the implementations of the goals described on this section.

3.1 Overview

Squirrel Core is divided in two main components: **Frontier** and **Worker**. The execution of Squirrel requires one frontier running and the user can set how many workers the system supports. Both modules uses the Spring framework for dependency injection of their modules. By that, the user can define which implementations will be used by changing a Spring beans configuration file, making it easier to select and include new modules on the run.

The Frontier is initialized by a list of input seeds. It will normalize and add all the identified URI's to a filter and to a queue. Once the Frontier receives a call from a Worker, will give all the URI's in the queue to the worker. The worker will only be initialized if there is a frontier available to connect to. Initially, it will request new URI's to crawl to the Frontier. Then, it will fetched data available from the URI, analyze the fetched data and store the content on the collector and the sink. The URI's found will then sent to the Frontier, which will check if the received URI's are on the filter and in case they are not, they are added to the queue, repeating the process until the queue is empty, meaning that there is no more URI's left to be crawled. Figure 1 illustrates the core architecture of Squirrel.

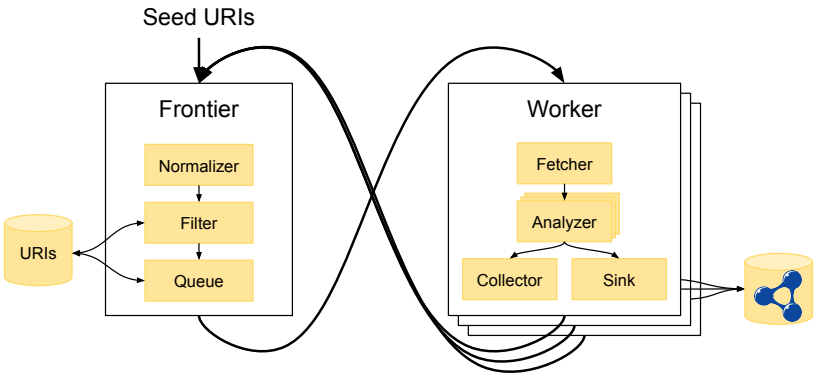

Fig. 1. Squirrel Core Achitecture

3.2 Frontier

The frontier has the task to organize the crawling. It keeps track of the URIs that should be crawled and those that already have been crawled. It comprises three main modules. A Normalizer that preprocesses incoming URIs, a Filter that filters them and a Queue which is used to keep track of the URIs that should be crawled in the future.

Normalizer. The Normalizer is preprocessing the incoming URIs by transforming them into a normal form. This mainly focusses on HTTP URIs and includes:

- Removal of default ports
- Replacement of unnecessary escaping
- Normalization of the URIs path
- Removal of the fraction
- Sorting of the query attributes

Note that the Normalizer won't change URIs that are already given in a normal form.

Filter. The Filter module is mainly responsible for filtering URIs that already have been processed. To this end, the Frontier makes use of a MongoDB instance which is used to store all these crawled URIs. Additionally, black- or whitefilters can be used to narrow the search space of the crawler if necessary. For each of the Modules used

Queue. The Queue is the module which takes the URIs that should be crawled. It groups, sorts and persists the URIs. Therefor, the implementation of the queue covers a major part of the crawling strategy the crawler uses. At the moment, Squirrel offers two queue implementations—a IP- as well as a URI-based queue. Both are working in a similar way by grouping URIs by their IP or their pay level domain, respectively. The URI groups are sorted following the FIFO principle. The persistence is achieved by storing the URI groups in MongoDB.

Recrawling. When the recrawling feature is activated, the Frontier checks the list of known URIs from time to time to retrieve URIs that have been crawled before but should be crawled again.

3.3 Worker

The worker component performs the crawling, which is done in four steps: 1) fetching the content of a URI, 2) analyzing the content, 3) collecting new URIs and 4) storing the content in a sink. The modules for these steps are described in the following.

Fetcher. The fetcher module takes the given URI and downloads its content. At the moment, Squirrel uses four different fetchers—two general fetchers for the HTTP and the FTP protocol as well as two fetchers which are focusing on SPARQL endpoints and CKAN portals, respectively. All these fetchers store the downloaded data in a temporary file on the disk of the worker. The crawler will try systematically all the fetchers loaded by Spring dependency injection. When it is finished, the fetcher module will store the file mime type on URI's properties, for further use by the analyzer module. In some cases, the HTTP fetcher may not get the mime type from the host, because it is absent. The fetcher module will save the mime type then, as unknown.

Analyzer. After the fetching phase, Squirrel checks if the fetched file is compressed, supporting the following formats: Gzip, Zip, Tar, 7z and Bzip2. If the file is compressed, it will be decompressed on the OS temp folder and each file found inside will be given to the Analyzer module.

The Analyzer module analyzes each downloaded file. Depending on the content type as well as the URI, a different Analyzer might be chosen. All the analyzers should implement the Analyzer interface and override the **analyze** and **isElegible** methods. The isElegible method, will check if that analyzer implementation is capable of dealing with the fetched data and if it is will call the analyze method. The analyze method, will receive the URI that is being crawled, the fetched file and the sink sik implementation chosen. In the current implementation, the following analyzers are available:

- RDF Analyzer for RDF files. The following serializations are supported by this analyzer:
  - RDF/XML
  - N-Triples, N3, NQ and N-Quads
  - Turtle
  - TTL
  - TRIG and TRIX
  - JsonLD
- RDFa Analyzer for HTML and XHTML Documents.
- HTML Scraper. An Analyzer for scrapping HTML pages. It uses the Jsoup framework [10] for scrapping and can be configured by the usage of yaml files to define how it should scrape a certain domain and its contexts or pages.
- Two fetchers are available for SPARQL endpoits. The **SparqlBasedFetcher** fetches all the triples from the host and the **SparqlDatasetFetcher** fetches data that uses the DCAT Ontology [1]
- The CKAN Analyzer is used for the JSON lines files which are loaded from the CKAN API. It transforms the information about datasets in the CKAN portal into RDF triples using the DCAT ontology [1]
- Any23-based analyzer that handles Microdata and Microformat.

[10] https://jsoup.org/

Collector. The Collector module is responsible for collecting all URIs that are gathered from the data. While a simple in-memory implementation has been developed, it is mainly used for testing. In case of analyzing large RDF datasets, the amount of new URIs is too large for the main memory and an SQL-based collector is used which persists the URIs in an SQL database. When the worker finished the crawling of a URI, it sends all the collected URIs to the Frontier.

Sink. The Sink has the task to persist extracted triples. At the moment, three sinks are implemented. The **FileSink** stores the triples locally in files. This file-based sink supports several RDF serializations and compresses the created files. Also, there is a **HDTBasedSink** available that stores the data on HDT compressed format [4]. Additionally, a **SPARQL-based sink** is available. This sink uses SPARQL queries to insert the new triples into a triple store.

Activity. The Worker implementation keeps track of the steps that are done to crawl a single URI. These steps as well as additional metadata are stored in the sink in a graph solely comprising metadata. The crawler is using the PROV ontology [11] to represent the crawling process as an activity which can have a graph in the sink as a result, as illustrated by the Figure 2. The activity tracks the number of triples found, which fetcher and analyzers were user, the starting and ending time, on which sink implementation was used, the ip address of the URI and the status, if it was successful or not.

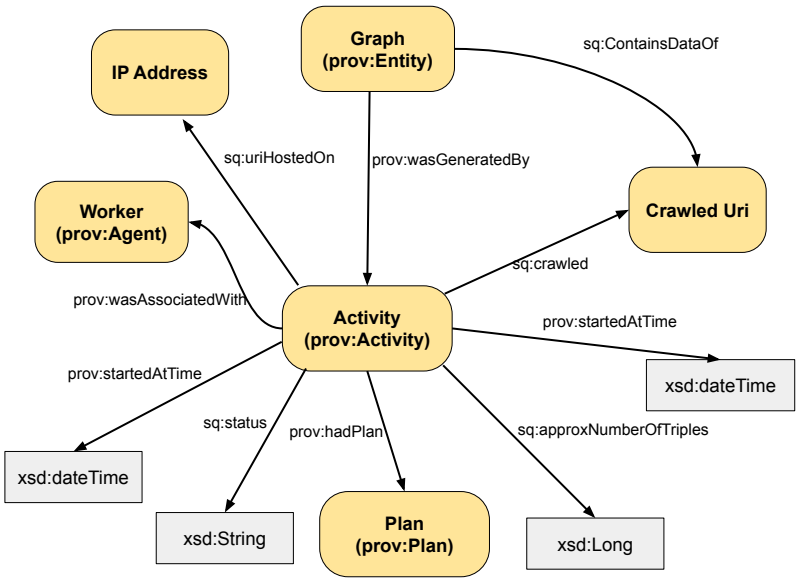

Fig. 2. Squirrel Activity, extending the PROV ontology

3.4 Triple store

Jena TDB and Virtuoso. In general, every triple store which supports SPARQL 1.1 can be used.

4 Application

Squirrel have been used for the Opal Project, an integrated portal for open data.[11] Opal integrates datasets from several datasources from Europe. Currently, Opal uses data from the following data portals: Mcloud.de, govdata.de and europeandataportal.eu. Also, it uses several sources found on OpenData-Monitor.eu, an indexer for several open data datasources.

Different strategies were used for the different data types. Mcloud does not has a Ckan or Sparql endpoint and dump links is available on each dataset catalog page. Because of it, the HttpFetcher and HTMLScraperAnalyzer were used to extract the dump links from HTML* pages. Ckan dataportals were extracted from OpenData Monitor using the same setup. In the future, SPARQL Types will be extracted as well. The config files that were used for HTMLScraper are available at: https://github.com/projekt-opal/squirrel-portals-config.

Europeandataportal was crawled using the SparqlDatasetFetcher and the RDFAnalyzer. The overall statistics from the crawling tasks are described on the table 2.

	Datasets	Triples	Runtime	Type
Mcloud.de	1 394	19 038	25min	HTML*,dump
govdata.de	34 057	138 669	4h	CKAN
europeandataportal.eu	1 008 379	13 404 005	36h	SPARQL
OpenDataMonitor	104 361	464 961	7h	CKAN

Table 2. Squirrel crawling statistics for Opal projekt.

5 Discussion

Squirrel has been developed with the variety of the linked web on mind. One of the advantages of the crawler, is the capability of dealing with several linked web protocols and RDF serializations. With the default modules, it is unlikely the crawler to be unsuccessful about harvesting all the data available from the linked web. The modular architecture and the use of the modules as spring beans, facilitates the development, implementation and inclusion of new modules, to satisfy the needs that the default modules may not fulfill.

[11] http://projekt-opal.de/

One of the important features that Squirrel has, is the metadata storage from the crawled URI's. This open possibilities for extracting insights from the crawling process, such as which URI's had failed to be crawled for a certain domain and the fetchers and analyzers used for the task, allowing the identification of problems, selecting better modules for a specific datasource or developing a module for it.

6 Conclusion

This paper presented Squirrel, a distributed linked data crawler that provides scalability and can handles several protocols and RDF serializations. We described the components and each module of the crawler and an use case, the Opal Projekt. Additionally, there is space for improvements in the future. The run time efficiency can be very low if all fetchers and analyzers are loaded. There is some bottlenecks that explain this deficiency. The worker does not know which fetcher is the most suitable for a given URI, since it is no possible to classify which type the URI it is, by only reading it. This does not include, of course, the exceptions of obvious incomes, such as URI's that have 'ckan' or 'sparql' spelled out. Consequently, the crawler will try all the fetchers systematically, causing the process to hang unnecessary on fetchers that are irrelevant.

The RDFAnalyzer will also be improved in the future. Like described on subsection 3.3 about the Fetchers, if the host does not provide the mime type of the file on the header, the HTTP fetcher will mark the mime type as unknown. Consequently, if the RDFAnalyzer is elegible for this URI, it will not know which RDF serialization should use. This will cause the analyzer to try all the possible serializations, causing an unnecessary delay. To increase the efficiency, due to this issues, it is necessary to increase the number of workers, which will consume more system resources.

The both situations described previosly, were identified during development and there is plans for future improvements in order to solve both bottlenecks. On future releases, the frontier will be improved to detect the correct URI type and the RDFAnalyzer will have a feature to detect the best RDF serialization for a given file. As described before, one of the goals of the crawler since the begginning, was to be able to handle the diversity of the linked web, so despite these two issues, this does not compromise it.

References

1. Archer, P.: Data catalog vocabulary (dcat) (w3c recommendation). Online (January 2014), https://www.w3.org/TR/vocab-dcat/
2. Beek, W., Rietveld, L., Bazoobandi, H.R., Wielemaker, J., Schlobach, S.: Lod laundromat: A uniform way of publishing other people's dirty data. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) The Semantic Web – ISWC 2014. pp. 213–228. Springer International Publishing, Cham (2014)

3. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: Lod-a-lot. In: d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C., Heflin, J. (eds.) The Semantic Web – ISWC 2017. pp. 75–83. Springer International Publishing, Cham (2017)
4. Fernndez, J.D., Martnez-Prieto, M.A., Gutirrez, C., Polleres, A., Arias, M.: Binary rdf representation for publication and exchange (hdt). Web Semantics: Science, Services and Agents on the World Wide Web **19**, 2241 (2013), http://www.websemanticsjournal.org/index.php/ps/article/view/328
5. Harth, A., Umbrich, J., Decker, S.: Multicrawler: A pipelined architecture for crawling and indexing semantic web data. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) The Semantic Web - ISWC 2006. pp. 258–271. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
6. Heydon, A., Najork, M.: Mercator: A scalable, extensible web crawler. Word Wide Web (1999)
7. Hogan, A.: Exploiting RDFS and OWL for Integrating Heterogeneous, Large-Scale, Linked Data Corpora (2011), http://aidanhogan.com/docs/thesis/
8. Hogan, A., Harth, A., Umbrich, J., Kinsella, S., Polleres, A., Decker, S.: Searching and browsing linked data with SWSE: The semantic web search engine. Web Semantics: Science, Services and Agents on the World Wide Web **9**(4), 365 – 401 (2011). https://doi.org/http://dx.doi.org/10.1016/j.websem.2011.06.004, http://www.sciencedirect.com/science/article/pii/S1570826811000473, JWS special issue on Semantic Search
9. Isele, R., Umbrich, J., Bizer, C., Harth, A.: LDspider: An open-source crawling framework for the Web of Linked Data. In: Proceedings of the ISWC 2010 Posters & Demonstrations Track: Collected Abstracts. vol. 658, pp. 29–32. CEUR-WS (2010)
10. Koster, M., Illyes, G., Zeller, H., Harvey, L.: Robots Exclusion Protocol. Internet-draft, Internet Engineering Task Force (IETF) (July 2019), https://tools.ietf.org/html/draft-rep-wg-topic-00
11. Lebo, T., Sahoo, S., McGuinness, D.: PROV-O: The PROV Ontology. W3C Recommendation, W3C (April 2013), http://www.w3.org/TR/2013/REC-prov-o-20130430/
12. Merkel, D.: Docker: Lightweight linux containers for consistent development and deployment. Linux J. **2014**(239) (Mar 2014), http://dl.acm.org/citation.cfm?id=2600239.2600241