Squirrel – Crawling RDF Knowledge Graphs on the Web

Michael Röder[1,2](✉), Geraldo de Souza Jr[1], and Axel-Cyrille Ngonga Ngomo[1,2]

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

Abstract. The increasing number of applications relying on knowledge graphs from the Web leads to a heightened need for crawlers to gather such data. Only a limited number of these frameworks are available, and they often come with severe limitations on the type of data they are able to crawl. Hence, they are not suited to certain scenarios of practical relevance. We address this drawback by presenting SQUIRREL, an open-source distributed crawler for the RDF knowledge graphs on the Web, which supports a wide range of RDF serializations and additional structured and semi-structured data formats. SQUIRREL is being used in the extension of national data portals in Germany and is available at https://github.com/dice-group/squirrel under a permissive open license.

Keywords: Linked data · Crawler · Open data

1 Introduction

The knowledge graphs available on the Web have been growing over recent years both in number and size [4][1]. This development has been accelerated by governments publishing public sector data on the web[2]. With the awareness of the power of 5-star linked open data has come the need for these organizations to (1) make the semantics of their datasets explicit and (2) connect their datasets with other datasets available on the Web. While the first step has been attended to in a plethora of projects on the semantification of data, the second goal has remained a challenge, addressed mostly manually. However, a manual approach to finding and linking datasets is impractical due to the steady growth of datasets provided by both governments and the public sector in both size and

[1] See https://lod-cloud.net/ for an example of the growth.
[2] Examples include the European Union at https://ec.europa.eu/digital-single-market/en/open-data and the German Federal Ministry of Transport and Digital Infrastructure with data at https://www.mcloud.de/.

J. Z. Pan et al. (Eds.): ISWC 2020, LNCS 12507, pp. 34–47, 2020.
https://doi.org/10.1007/978-3-030-62466-8_3

number[3]. Different public services have hence invested millions of Euros into research projects aiming to automate the connection of government data with other data sources[4].

An indispensable step towards automating the second goal is the *automated and periodic gathering of information about available open data that can be used for linking to newly published data of the public sector*. A necessary technical solution towards this end is a *scalable crawler for the Web of Data*. While the need for such a solution is already dire, it will become even more pressing to manage the growing amount of data that will be made available each year into the future. At present, the number of open-source crawlers for the web of data that can be used for this task is rather small and all come with several limitations. We close this gap by presenting Squirrel—a distributed, open-source crawler for the web of data[5]. Squirrel supports a wide range of RDF serializations, decompression algorithms and formats of structured data. The crawler is designed to use Docker[6] containers to provide a simple build and run architecture [13]. Squirrel is built using a modular architecture and is based on the concept of dependency injection. This allows for a further extension of the crawler and adaptation to different use cases.

The rest of this paper is structured as follows: we describe related work in Sect. 2 and the proposed crawler in Sect. 3. Section 4 presents an evaluation of the crawler, while Sect. 5 describes several applications of Squirrel. We conclude the paper in Sect. 6.

2 Related Work

There are only a small number of open-source Data Web crawlers available that can be used to crawl RDF datasets. An open-source Linked Data crawler to crawl data from the web is LDSpider[7] [10]. It can make use of several threads in parallel to improve the crawling speed, and 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. The crawled data can be stored either in files or can be sent to a SPARQL endpoint. It supports a limited amount of RDF serialisations (details can be found in Table 1 in Sect. 3). In addition, it cannot be deployed in a distributed environment. Another limitation of LDSpider is the missing functionality to crawl SPARQL endpoints and open data portals. A detailed comparison of LDSpider and Squirrel can be found in Sects. 3 and 4.

[3] See, e.g., https://www.mdm-portal.de/, where traffic data from the German Federal Ministry of Transport and Digital Infrastructure is made available.

[4] See, e.g., the German mFund funds at http://mfund.de.

[5] Our code is available at https://github.com/dice-group/squirrel and the documentation at https://w3id.org/dice-research/squirrel/documentation.

[6] https://www.docker.com/.

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

A crawler focusing on structured data is presented in [6]. The authors describe a 5-step pipeline that converts structured data formats like XHTML or RSS into RDF. In [8,9], a distributed crawler is described, which is used to index resources for the Semantic Web Search Engine. To the best of our knowledge, both crawlers are not available as open-source projects.

In [2], the authors present the LOD Laundromat—a framework that downloads, parses, cleans, analyses and republishes RDF datasets. The framework has the advantage of coming with a robust parsing algorithm for various RDF serialisations. However, it solely relies on a given list of seed URLs. In contrast to a crawler, it does not extract new URLs from the fetched data to crawl.

Since web crawling is an established technique, there are several open-source crawlers. An example of a scalable, general web crawler is presented in [7]. However, most of these crawlers cannot process RDF data without further adaptation. A web crawler extended for processing RDF data is the open-source crawler Apache Nutch[8]. Table 1 in Sect. 3 shows the RDF serialisations, compressions and forms of structured data that are supported by the Apache Nutch plugin[9]. However, the plugin stems from 2007, relies on an out-dated crawler version and failed to work during our tests[10].

Overall, the open-source crawlers currently available are either not able to process RDF data, are limited in the types of data formats they can process, or are restricted in their scalability.

3 Approach

3.1 Requirements

Web of Data crawler requirements were gathered from nine organisations within the scope of the projects LIMBO[11] and OPAL[12]. OPAL aims to create an open data portal by integrating the available open data of different national and international data sources[13]. The goal of LIMBO is to collect available mobility data of the ministry of transport, link them to open knowledge bases and publish them within a data portal[14].

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

R1: The crawler should be designed to provide a distributed and scalable solution on crawling structured and semi-structured data.

[8] http://nutch.apache.org/.

[9] The information has been gathered by an analysis of the plugin's source code.

[10] A brief description of the plugin and its source code can be found at https://issues.apache.org/jira/browse/NUTCH-460.

[11] https://www.limbo-project.org/.

[12] http://projekt-opal.de/projektergebnisse/deliverables/.

[13] See http://projekt-opal.de/en/welcome-project-opal/ and https://www.bmvi.de/SharedDocs/DE/Artikel/DG/mfund-projekte/ope-data-portal-germany-opal.html.

[14] See https://www.limbo-project.org/ and https://www.bmvi.de/SharedDocs/DE/Artikel/DG/mfund-projekte/linked-data-services-for-mobility-limbo.html.

R2: The crawler must exhibit "respectful" behaviour when fetching data from servers by following the Robots Exclusion Standard Protocol [11]. This reduces the chance that a server is overloaded by the crawler's request and the chance that the crawler is blocked by a server.
R3: Since not all data is available as structured data, crawlers for the data web should offer a way to gather semi-structured data.
R4: The project should offer easy addition of further functionality (e.g., novel serialisations, other types of data, etc.) through a fully extensible architecture.
R5: The crawler should provide metadata about the crawling process, allowing users to get insights from the crawled data.

In the following, we give an overview of the crawler's components, before describing them in more detail.

3.2 Overview

Squirrel comprises two main components: *Frontier* and *Worker* (**R1**). To achieve a fully extensible architecture, both components rely on the dependency injection pattern, i.e., they comprise several modules that implement the single functionalities of the components. These modules can be injected into the components, facilitating the addition of more functionalities (**R4**). To support the addition of the dependency injection, Squirrel has been implemented based on the Spring framework[15]. Fig. 1 illustrates the architecture of Squirrel.

When executed, the crawler has exactly one Frontier and a number of Workers, which can operate in parallel (**R1**). The Frontier is initialised with a list of seed URIs. It normalises and filters the URIs, which includes a check of whether the URIs have been seen before. Thereafter, the URIs are added to the internal queue. Once the Frontier receives a request from a Worker, it gives a set of URIs to the Worker. For each given URI, the Worker fetches the URI's content, analyses the received data, collects new URIs and forwards the data to its sink. When the Worker is done with the given set of URIs, it sends it back to the Frontier together with the newly identified URIs. The crawler implements the means for a periodic re-evaluation of URIs known to have been crawled in past iterations.

3.3 Frontier

The Frontier has the task of organising the crawling. It keeps track of the URIs to be crawled, and those that have already been crawled. It comprises three main modules:

1. A Normalizer that preprocesses incoming URIs,
2. a Filter that removes already seen URIs
3. a Queue used to keep track of the URIs to be crawled in the future.

[15] https://spring.io/.

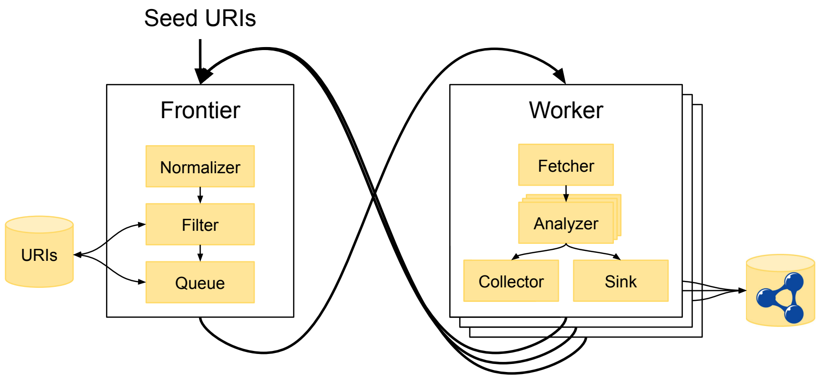

Fig. 1. Squirrel core achitecture

3.3.1 Normalizer

The Normalizer preprocesses incoming URIs by transforming them into a normal form. This reduces the number of URIs that are different but point to the same resources. The URI normalisation comprises the following actions:

– Removal of default ports, e.g., port 80 for HTTP.
– Removal of percentage-encoding for unreserved characters [3].
– Normalization of the URI path, e.g., by removing punctuations [3].
– Removal of the URIs' fragment part.
– Alphanumeric sorting of key-value pairs for query parts that contain several key value pairs.

In addition, the Normalizer tries to find session identifiers or similar parts of the URI that have no influence on the retrieved content. The strings that mark such a part of the URI are configurable.

3.3.2 Filter

The Filter module is mainly responsible for filtering URIs that have already been processed. To achieve this goal, the Frontier makes use of a NoSQL database (i.e., MongoDB in the current implementation), which is used to store all crawled URIs in a persistent way. This ensures that a crawler can be interrupted and restarted later on. Additionally, black or white lists can be used to narrow the search space of the crawler if necessary.

3.3.3 Queue

The Queue is the module that stores the URIs to be crawled. It groups and sorts the URIs, which makes it the main module for implementing crawling strategies. At present, Squirrel offers two queue implementations—an IP- and a domain-based first-in-first-out (short: FIFO) queue. Both work in a similar way by grouping URIs based on their IP or their pay-level domain, respectively. The

URI groups are sorted following the FIFO principle. When a Worker requests a new set of URIs, the next available group is retrieved from the queue and sent to the Worker. Internally, this group is marked as blocked, i.e., it remains in the queue and new URIs can be added by the Frontier but it cannot be sent to a different Worker. As soon as the Worker returns the requested URIs, the group is unblocked and the crawled URIs are removed from it. If the group is empty, it is removed from the queue. This implements a load-balancing strategy that aims to crawl the web as fast as possible without overloading single IPs or pay-level domains.

Like the Filter module, the Queue relies on a persistent MongoDB to store the URIs. This enables a restart of the Frontier without a loss of its internal states.

3.4 Worker

The Worker component performs the crawling based on a given set of URIs. Crawling a single URI is done in four steps:

1. URI content is fetched,
2. fetched content is analysed,
3. new URIs are collected, and
4. the content is stored in a sink.

The modules for these steps are described in the following:

3.4.1 Fetcher

The fetcher module takes a given URI and downloads its content. Before accessing the given URI, the crawler follows the Robots Exclusion Standard Protocol [11] and checks the server's `robots.txt` file (**R2**). If the URI's resource can be crawled, one of the available fetchers is used to access it. At present, Squirrel uses four different fetchers. Two general fetchers cover the HTTP and the FTP protocol, respectively. Two additional fetchers are used for SPARQL endpoints and CKAN portals, respectively. However, other fetchers can be added by means of the extensible Squirrel API if necessary[16].

The Worker tries to retrieve the content of the URI by using the fetchers, in the order in which they were defined, until one of them is successful. The fetcher then stores the data on the disk and adds additional information (like the file's MIME type) to the URI's properties for later usage. Based on the MIME type, the Worker checks whether the file is a compressed or an archive file format. In this case, the file is decompressed and extracted for further processing. In its current release, Squirrel supports the formats Gzip, Zip, Tar, 7z and Bzip2[17].

[16] Details about implementing a new fetcher can be found at https://dice-group.github.io/squirrel.github.io/tutorials/fetcher.html.

[17] Details regarding the compressions can be found at https://pkware.cachefly.net/Webdocs/APPNOTE/APPNOTE-6.3.5.TXT, https://www.gnu.org/software/gzip/ and http://sourceware.org/bzip2/, respectively.

Table 1. Comparison of RDF serialisations, compressions, methods to extract data from HTML and other methods to access data supported by Apache Nutch (including the RDF plugin), LDSpider and Squirrel.

	RDF Serialisations										Comp.					HTML					
	RDF/XML	RDF/JSON	Turtle	N-Triples	N-Quads	Notation 3	JSON-LD	TriG	TriX	HDT	ZIP	Gzip	bzip2	7zip	tar	RDFa	Microdata	Microformat	HTML (scraping)	SPARQL	CKAN
Apache Nutch	✓	–	✓	✓	–	✓	–	–	–	–	✓	✓	–	–	–	✓	✓	✓	–	–	–
LDSpider	✓	–	✓	✓	✓	✓	✓	–	–	–	–	–	–	–	–	✓	✓	✓	–	–	–
Squirrel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

3.4.2 Analyser

The task of the Analyser module is to process the content of the fetched file and extract triples from it. The Worker has a set of Analysers that are able to handle various types of files. Table 1 lists the supported RDF serialisations, the compression formats and the different ways Squirrel can extract data from HTML files. It compares the supported formats with the formats supported by Apache Nutch and LDSpider [10]. Each Analyser offers an `isElegible` method that is called with a URI and the URI's properties to determine whether it is capable of analysing the fetched data. The first Analyser that returns true receives the file together with a Sink and a Collector, and starts to analyse the data.

The following Analysers are available in the current implementation of Squirrel:

1. The RDF Analyser handles RDF files and is mainly based on the Apache Jena project[18]. Thus, it supports the following formats: RDF/XML, N-Triples, N3, N-Quads, Turtle, TRIG, JSON-LD and RDF/JSON.
2. The HDT Analyser is able to process compressed RDF graphs that are available in the HDT file format [5].
3. The RDFa Analyser processes HTML and XHTML Documents extracting RDFa data using the Semargl parser[19].
4. The scraping Analyser uses the Jsoup framework for parsing HTML pages and relies on user-defined rules to extract triples from the parsed page[20].

[18] https://jena.apache.org.
[19] https://github.com/semarglproject/semargl.
[20] https://jsoup.org/.

This enables the user to use Squirrel to gather not only structured but also semi-structured data from the web (**R3**).
5. The CKAN Analyser is used for the JSON line files generated by the CKAN Fetcher when interacting with the API of a CKAN portal. The analyser transforms the information about datasets in the CKAN portal into RDF triples using the DCAT ontology [1].
6. The Any23-based Analyser processes HTML pages, searching for Microdata or Microformat embedded within the page.
7. In contrast to the other Fetchers, the SPARQL-based Fetcher directly performs an analysis of the retrieved triples.

New analysers can be implemented if the default API does not match the user's needs[21].

3.4.3 Collector

The Collector module collects all URIs from the RDF data. Squirrel offers an SQL-based collector that makes use of a database to store all collected URIs. It ensures the scalability of this module for processing large data dumps. For testing purposes, a simple in-memory collector is provided. As soon as the Worker has finished crawling the given set of URIs, it sends all collected URIs to the Frontier and cleans up the collector.

3.4.4 Sink

The Sink has the task to store the crawled data. Currently, a user can choose from three different sinks that are implemented. First, a file-based sink is available. This sink stores given triples in files using the Turtle serialisation for RDF[22]. These files can be further compressed using GZip. The second sink is an extension of the file-based sink and stores triples in the compressed HDT format [5]. It should be noted that both sinks separate the crawled data by creating one file for each URI that is crawled. An additional file is used to store metadata from the crawling process. Both sinks have the disadvantage that each Worker has a local directory in which the data is stored. The third sink uses SPARQL update queries to insert the data in a SPARQL store. This store can be used by several Workers in parallel. For each crawled URI, a graph is created. Additionally, a metadata graph is used to store the metadata generated by the Workers. New sinks can be added by making use of the extensible API[23].

3.4.5 Activity

The Workers of Squirrel document the crawling process by writing metadata to a metadata graph (**R5**). This metadata mainly relies on the PROV ontol-

[21] Details about implementing a new analyzer can be found at https://dice-group.github.io/squirrel.github.io/tutorials/analyzer.html.

[22] https://www.w3.org/TR/turtle/.

[23] Details about implementing a new sink can be found at https://dice-group.github.io/squirrel.github.io/tutorials/sink.html.

ogy [12] and has been extended where necessary. Figure 2 gives an overview of the generated metadata. The crawling of a single URI is modelled as an activity. Such an activity comes with data like the start and end time, the approximate number of triples received, and a status line indicating whether the crawling was successful. The result graph (or the result file in case of a file-based sink) is an entity generated by the activity. Both the result graph and the activity are connected to the URI that has been crawled.

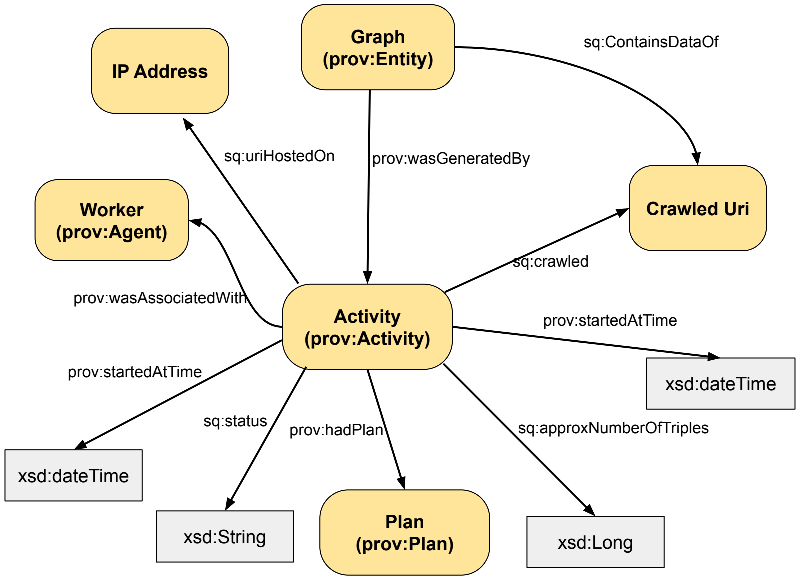

Fig. 2. Squirrel activity, extending the PROV ontology

4 Evaluation

4.1 Benchmark

We carried out two experiments to compare Squirrel with the state-of-the-art Data Web crawler, LDSpider [10]. LDSpider was chosen because it is one of the most popular crawlers for the linked web, and is widely used in various projects. All experiments were carried out using the ORCA benchmark for Data Web crawlers [15][24]. ORCA is built upon the HOBBIT benchmarking platform [14] and ensures repeatable and comparable benchmarking of crawlers for the Data Web. To this end, it creates a network of servers from which the crawler should download data. For each server, ORCA generates an RDF dataset with outgoing

[24] https://github.com/dice-group/orca.

links to other servers. This allows the crawler to start with one or more seed URIs and crawl the complete network. Note, the benchmark ensures that all triples created for the single servers can be reached by a crawler by traversing the links. ORCA offers five different types of servers:

1. a dump file server,
2. a SPARQL endpoint,
3. a CKAN portal,
4. a server for HTML with embedded RDFa triples
5. a dereferencing server.

The latter can be called via the URI of an RDF resource and answers with the triples that have the resource as subject. The dereferencing server negotiates the RDF serialisation with the crawler based on the crawler's HTTP request. However, the serialisation of each dump file is randomly chosen to be either RDF/XML, Turtle, N-Triples or Notation 3. ORCA measures the completeness of data gathered by the crawler, and its run time.

4.2 Evaluation Setup

We carry out two experiments in which we use ORCA to simulate a network of servers. The first experiment simulates a real-world Data Web and focuses on the effectiveness of the two crawlers, i.e., the amount of correct triples they retrieve. As suggested by the authors of [15], the generated cloud comprises 100 servers with 40% dump file servers, 30% SPARQL servers, 21% dereferencing servers, 5% CKAN servers and 4% servers that offer RDFa within HTML pages. The average degree of each node is set to 20 and the data of each node comprises 1000 triples with an average degree of 9 triples per resource. 30% of the dump file nodes offer their files compressed using zip, gzip or bz2. The results of this experiment are listed in Table 2[25].

The second experiment focuses on the efficiency of the crawler implementations. We follow the suggestion given in [15] for efficiency experiments and solely rely on 200 dereferencing servers. These servers have can negotiate the RDF serialisation with the crawler. Hence, the crawlers are very likely to be able to crawl the complete graph, which eases a comparison of the crawlers with respect to their efficiency. The other parameters are the same as in the first experiment. The results of the second experiment are listed in Table 2[26].

For all experiments, we use a cluster of machines. The crawlers are deployed on 3 machines while the created servers of the ORCA benchmark are hosted on 3 other machines. Each of the machines has 16 cores with hyperthreading

[25] The detailed results can be seen at https://w3id.org/hobbit/experiments#1585403645660,1584545072279,1585230107697,1584962226404,1584962243223,1585574894994,1585574924888,1585532668155,1585574716469.

[26] Detailed results can be found at https://w3id.org/hobbit/experiments#1586886425879,1587151926893,1587284972402,1588111671515,1587121394160,1586886364444,1586424067908,1586374166710,1586374133562.

Table 2. Results for experiment I and II.

Crawler	Experiment I		Experiment II			
	Micro Recall	**Run time** (in s)	**Micro Recall**	**Run time** (in s)	**CPU** (in s)	**RAM** (in GB)
LDSpider (T1)	0.31	1798	1.00	2 031	320.0	1.2
LDSpider (T8)	0.30	1792	1.00	2 295	265.9	2.8
LDSpider (T16)	0.31	1858	1.00	1 945	345.4	1.6
LDSpider (T32)	0.31	1847	1.00	2 635	11.6	2.6
LDSpider (T32,LBS)	0.03	66	0.54	765	182.1	7.5
Squirrel (W1)	0.98	6 663	1.00	11 821	991.3	3.9
Squirrel (W3)	0.98	2 686	1.00	4 100	681.4	8.6
Squirrel (W9)	0.98	1 412	1.00	1 591	464.8	18.1
Squirrel (W18)	0.97	1 551	1.00	1 091	279.8	22.1

and 256 GB RAM[27]. For both experiments, the usage of `robots.txt` files is disabled. We use several configurations of LDSpider and Squirrel. LDSpider (T1), (T8), (T16) and (T32) use a breadth-first crawling strategy and 1, 8, 16 or 32 threads, respectively. Additionally, we configure LDSpider (T32,LSB), which makes use of 32 threads and a load-balancing strategy. Further, we configure Squirrel (W1), (W3), (W9) and (W18) to use 1, 3, 9 or 18 Worker instances, respectively.

4.3 Discussion

The results of the first experiment show that LDSpider has a lower recall than Squirrel. This difference is due to several factors. LDSpider does not support 1) the crawling of SPARQL endpoints, 2) the crawling of CKAN portals, nor 3) the processing of compressed RDF dump files. In comparison, Squirrel comes with a larger set of supported server types, RDF serialisations and compression algorithms. Hence, Squirrel was able to crawl nearly all triples. However, not all triples of all CKAN portals and RDFa nodes could be retrieved, leading to a micro recall of up to 0.98.

The second experiment shows that the larger set of features offered by Squirrel comes with lower efficiency. LDSpider achieves lower run times using a more economical configuration with respect to consumed CPU time and RAM. With a higher number of workers, Squirrel achieves lower run times but consumes much more RAM than LDSpider. At the same time, the experiment reveals that the load-balancing strategy of LDSpider tends to abort the crawling process very early and, hence, achieves only a low recall in both experiments.

[27] The details of the hardware setup that underlies the HOBBIT platform can be found at https://hobbit-project.github.io/master#hardware-of-the-cluster.

5 Application

Squirrel is used within several research projects, of which two are of national importance in Germany. The OPAL project creates an integrated portal for open data by integrating datasets from several data sources from all over Europe[28]. At the moment, the project focuses on the portals `mCLOUD.de`, `govdata.de` and `europeandataportal.eu`. In addition, several sources found on `OpenDataMonitor.eu` are integrated. SQUIRREL is used to regularly gather information about available datasets from these portals. Table 3 lists the number of datasets that are extracted from the portals, the time the crawler needs to gather them, and the way the crawler accesses data. It should be noted that the run times include the delays SQUIRREL inserts between single requests to ensure that the single portals are not stressed. The portals evidently use different ways to offer their data. Two of them are CKAN portals, while `mCLOUD.de` has to be scraped using SQUIRREL's HTML scraper. Only `europeandataportal.eu` offers a SPARQL endpoint to access the dataset's metadata. The data integrated by OPAL are to be written back into the `mCLOUD.de` portal and cater for the needs of private and public organisations requiring mobility data. Users range from large logistic companies needing to plan transport of goods, to single persons mapping their movement with pollen concentration.

Table 3. Crawling statistics of the OPAL project.

	Datasets	Triples	Run time	Type
`mCLOUD.de`	1 394	19 038	25min	HTML
`govdata.de`	34 057	138 669	4h	CKAN
`europeandataportal.eu`	1 008 379	13 404 005	36h	SPARQL
`OpenDataMonitor.eu`	104 361	464 961	7h	CKAN

Another project that makes use of SQUIRREL to collect data from the web is LIMBO. Its aim is to unify and refine mobility data of the German Federal Ministry of Transport and Digital Infrastructure. The refined data is made available to the general public to create the basis for new, innovative applications. To this end, SQUIRREL is used to collect this and related data from different sources.

6 Conclusion

This paper presented SQUIRREL, a scalable, distributed and extendable crawler for the Data Web, which provides support for several different protocols and data serialisations. Other open-source crawlers currently available are either not able to process RDF data, are limited in the types of data formats they can process, or are restricted in their scalability. SQUIRREL addresses these drawbacks

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

by providing an extensible architecture adaptable to supporting any format of choice. Moreover, the framework was implemented for simple deployment both locally and at a large scale.

We described the components of the crawler and presented a comparison with LDSpider. This comparison showed the advantages of Squirrel with respect to the large amount of supported data and server types. Squirrel was able to crawl data from different sources (HTTP, SPARQL and CKAN) and compression formats (zip,gzip,bz2), leading to a higher recall than LDSpider. In addition, we identified Squirrel's efficiency as a focus for future development and improvement. Squirrel is already used by several projects and we provide tutorials for its usage to empower more people to make use of the data available on the web[29].

Acknowledgments. This work has been supported by the BMVI (Bundesministerium für Verkehr und digitale Infrastruktur) projects LIMBO (GA no. 19F2029C) and OPAL (GA no. 19F2028A).

References

1. Archer, P.: Data catalog vocabulary (dcat) (w3c recommendation), 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., et al. (eds.) The Semantic Web - ISWC 2014, pp. 213–228. Springer International Publishing, Cham (2014)
3. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. Internet Standard, Internet Engineering Task Force (IETF), January 2005. https://tools.ietf.org/html/rfc3986
4. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: LOD-a-lot. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 75–83. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_7
5. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semant. Sci. Serv. Agents World Wide Web, **19**, 22–41 (2013). http://www.websemanticsjournal.org/index.php/ps/article/view/328
6. Harth, A., Umbrich, J., Decker, S.: MultiCrawler: a pipelined architecture for crawling and indexing semantic web data. In: Cruz, I., et al. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 258–271. Springer, Heidelberg (2006). https://doi.org/10.1007/11926078_19
7. Heydon, A., Najork, M.: Mercator: a scalable, extensible web crawler. Word Wide Web **2**(4), 219–229 (1999)
8. Hogan, A.: Exploiting RDFS and OWL for Integrating Heterogeneous, Large-Scale, Linked Data Corpora (2011). http://aidanhogan.com/docs/thesis/

[29] https://dice-group.github.io/squirrel.github.io/tutorials.html.

9. 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 Semant. Sci. Serv. Agents World Wide Web, **9**(4), 365–401 (2011). https://doi.org/10.1016/j.websem.2011.06.004. http://www.sciencedirect.com/science/article/pii/S1570826811000473, JWS special issue on Semantic Search
10. 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)
11. 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
12. 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/
13. Merkel, D.: Docker: Lightweight linux containers for consistent development and deployment. Linux J. 2014(239), March 2014. http://dl.acm.org/citation.cfm?id=2600239.2600241
14. Röder, M., Kuchelev, D., Ngonga Ngomo, A.C.: HOBBIT: a platform for benchmarking Big Linked Data. Data Sci. (2019). https://doi.org/10.3233/DS-190021
15. Röder, M., de Souza, G., Kuchelev, D., Desouki, A.A., Ngomo, A.C.N.: Orca: a benchmark for data web crawlers (2019). https://arxiv.org/abs/1912.08026