PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Raphael do Vale Amaral Gomes

Crawling the Linked Data Cloud

Tese de Doutorado

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Prof. Marco Antonio Casanova

Rio de Janeiro
May 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Raphael do Vale Amaral Gomes

Crawling the Linked Data Cloud

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor.

Prof. Marco Antonio Casanova
Advisor
Departamento de Informática – PUC-Rio

Prof. Antonio Luz Furtado
Departamento de Informática – PUC-Rio

Prof. Daniel Schwabe
Departamento de Informática – PUC-Rio

Profa. Giseli Rabello Lopes
UFRJ

Prof. Alberto Henrique Frande Laender
UFMG

Prof. Geraldo Bonorino Xexéo
UFRJ

Prof. José Eugênio Leal
Coordinator of the Centro Técnico Científico da PUC-Rio

Rio de Janeiro, May 12th, 2015

Raphael do Vale Amaral Gomes

Graduated in Computer Science at the Pontifical University of Rio de Janeiro in 2006 and obtained her M.Sc. Degree in Computer Science from the Pontifical University of Rio de Janeiro in 2010.

Bibliographic data

Gomes, Raphael do Vale Amaral

Crawling the Linked Data Cloud / Raphael do Vale Amaral Gomes ; advisor: Marco Antonio Casanova. – 2015.

118 f. : il. ; 30 cm

Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.

Inclui bibliografia

1. Informática – Teses. 2. Buscadores Focados. 3. Recomendação de triplesets. 4. Linked Data. I. Casanova, Marco Antonio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CCD: 004

To my girls.

Acknowledgements

Foremost, I would like to thank my advisor Prof. Marco Antonio Casanova for the continuous support during my PhD studies and research, for his patience, motivation, enthusiasm, friendship, and immense knowledge. I truly could not imagine being able to finish this work without his help.

Besides my advisor, I would like to thank two other researchers in special: Prof. Giseli Rabello Lopes and Prof. Luiz André P. Paes Leme. Their help, knowledge and friendship contributed to many parts of this work.

My sincere thanks to Prof. Luiz Fernando Bessa Seibel for introducing me the academy.

I would also like to thank my parents, Walter do Amaral Gomes Filho and Maria de Fatima do Vale Gomes, and my brother, Gabriel do Vale Amaral Gomes, for supporting me and giving me the strength to conclude this research.

Finally, I would like to thank my wife, Mariana Ribeiro do Vale for all the support during those years that we have been together, specially these last ones. Her patience, work, help and love certainly made this work possible.

Last but not least, I would like to thank my daughter, Lara Ribeiro do Vale, for giving me a new direction and a new sense in life.

Abstract

Gomes, Raphael do Vale Amaral; Casanova, Marco Antonio (Advisor). **Crawling the Linked Data Cloud.** Rio de Janeiro, 2015. 118p. D.Sc. Thesis - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The Linked Data best practices recommend to publish a new tripleset using well-known ontologies and to interlink the new tripleset with other triplesets. However, both are difficult tasks. This thesis describes frameworks for metadata crawlers that help selecting the ontologies and triplesets to be used, respectively, in the publication and the interlinking processes. Briefly, the publisher of a new tripleset first selects a set of terms that describe the application domain of interest. Then, he submits the set of terms to a metadata crawler, constructed using one of the frameworks described in the thesis, that searches for triplesets which vocabularies include terms direct or transitively related to those in the initial set of terms. The crawler returns a list of ontologies that are used for publishing the new tripleset, as well as a list of triplesets with which the new tripleset can be interlinked. Hence, the crawler focuses on specific metadata properties, including *subclass of*, and returns only metadata, which justifies the classification "metadata focused crawler".

Keywords

Focused Crawler; Tripleset Recommendation; Linked Data.

Resumo

Gomes, Raphael do Vale Amaral; Casanova, Marco Antonio (Orientador). **Coleta de Dados Interligados.** Rio de Janeiro, 2015. 118p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

As melhores práticas de dados interligados recomendam que se utilizem ontologias bem conhecidas de modo a facilitar a ligação entre um novo conjunto de triplas RDF (ou, abreviadamente, *tripleset*) e os já existentes. Entretanto, ambas as tarefas apresentam dificuldades. Esta tese apresenta frameworks para criação de buscadores de metadados que ajudam na seleção de ontologias e na escolha de triplesets que podem ser usados, respectivamente, nos processos de publicação e interligação de triplesets. Resumidamente, o administrador de um novo tripleset deve inicialmente definir um conjunto de termos que descrevam o domínio de interesse do tripleset. Um buscador de metadados, construído segundo os frameworks apresentados na tese, irá localizar, nos vocabulários dos triplesets existentes, aqueles que possuem relação direta ou indireta com os termos definidos pelo administrador. O buscador retornará então uma lista de ontologias que podem ser utilizadas para o domínio, bem como uma lista dos triplesets relacionados. O buscador tem então como foco os metadados dos triplesets, incluindo informações de subclasse, e a sua saída retorna somente metadados, justificando assim chama-lo de "buscador focado em metadados".

Palavras-chave

Buscadores Focados; Recomendação de triplesets; Linked Data.

Table of Contents

List of Figures

List of Tables

1
Introduction

1.1.
Motivation

The Linked Data best practices (Bizer et al., 2009) recommend publishers of triplesets to use well-known ontologies in the triplication process and to link their triplesets with other triplesets. However, despite the fact that extensive catalogues of open ontologies and triplesets are available, such as DataHub[1], most publishers typically do not adopt ontologies already in use. They usually link their triplesets only with popular ones, such as DBpedia[2] and Geonames[3]. Indeed, according to Nikolov and Martínez-Romero (Nikolov and d'Aquin, 2011; Nikolov et al. 2012; Martínez-Romero, 2010), linkage to popular triplesets is favored for two main reasons: the difficulty of finding related open triplesets; and the strenuous task of discovering instance mappings between different triplesets.

This thesis describes three crawlers that address the problem of finding vocabulary terms and triplesets to assist publishers in the triplification and the linkage processes. Unlike typical Linked Data crawlers, the proposed crawlers focus on metadata with specific purposes, illustrated in what follows.

In a typical scenario, the publisher of a tripleset first selects a set *T* of terms that describe an application domain. Alternatively, he could use a database summarization technique (Saint-Paul et al., 2005) to automatically extract *T* from a set of triplesets.

Then, the publisher submits *T* to the crawler, that will search for tripleset which vocabularies include terms direct or transitively related to those in *T* by, for example, the "*subset of*" metadata relationship. The crawler returns a list of terms and triplesets, as well as provenance data indicating how the output was generated.

[1] http://datahub.io

[2] http://dbpedia.org

[3] http://www.geonames.org

For example, if the publisher selects the term "*Music*" from WordNet, the crawler might return the term "*Hit music*" and might indicate that "*BBC Music*" is a tripleset where "*Hit music*" occurs.

Lastly, the publisher inspects the list of terms and triplesets returned, with respect to his tripleset, to select the most relevant vocabularies for the triplification process and the best triplesets to use in the linkage process, possibly with the help of recommender tools. We stress that the crawler was designed to help recommender tools for Linked Data, and not to replace them.

1.2. Contributions

This thesis proposes a new way to crawl metadata from the Linked Data cloud. By adopting SPARQL *crawling queries*, coupled with a breadth-first strategy, we are able to create crawlers that are capable of finding a new terms and triplesets related to an initial set of terms, without losing precision, when compared to other crawling tools.

In more detail, the first contribution of this thesis is a strategy to crawl metadata from the Linked Data cloud. Unlike other crawling strategies, we do not just follow links from one tripleset to the other, but we discover which terms and triplesets are semantically related to an initial set of terms by, for example, the "*subset of*" metadata relationship. The crawling strategy relies on SPARQL crawling queries to discover the new terms and triplesets: each crawling query captures a specific metadata relationship or just counts the number of instances of a class.

The second contribution is to prototype metadata crawling tools that implement the crawling strategy. By adopting SPARQL crawling queries, we simplify the metadata crawling process, since the tools do not require to store all the data they need, thereby reducing the amount of data processed. We implemented three tools with increasingly sophisticated architecture.

The third contribution is to model the metadata crawling tools as frameworks so that anyone is able to plug in new crawling techniques. Indeed, it will help other researchers to use tools already implemented to traverse the Linked Data cloud, with minimal development effort.

The final contribution of this thesis is the use of the Actor Model to create a crawling tool, with improved performance, that explores distributed computing throughout the crawling process. Despite the changes, all other contributions were preserved when moving to the Actor Model.

Finally, we remark that Chapter 5 reflects a paper published in 2014 in the *16th International Conference on Enterprise Information Systems (ICEIS)* (Gomes et al., 2014), which won the Best Paper Award in the area of Software Agents and Internet Computing. Chapter 6 is the result of a second paper published in the *16h International Conference, ICEIS 2014, Revised Selected Papers Series: Lecture Notes in Business Information Processing* (Gomes et al., 2015). Finally, Chapter 7, describes the Actor-based implementation, which be submitted for publication.

1.3. Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 presents a brief description of common concepts used in the rest of the thesis. Chapter 3 describes the metadata crawling strategy proposed and introduces a use case that will be adopted in the remaining chapters. Chapter 4 introduces a proof-of-concept implementation of the crawling strategy to test its adequacy. Chapter 5 details an optimized implementation, re-engineered as a crawling framework. Chapter 6 presents our last implementation, a crawling framework that uses the Actor Model (Hewitt et al., 1973) to address issues related to performance and distribution. The optimized implementation is called *CrawlerLD* and the Actor Model-based implementation is called *DIST-CrawlerLD*. Chapter 7 contains the conclusions and outlines suggestions for future work.

2
Background

2.1.
Linked Data Concepts and Tools

This section briefly reviews basic Linked Data concepts and tools that will be used throughout the thesis.

The Linked Data principles advocate the use of RDF (Manola et al., 2004), RDF Schema (Brickley at al., 2004) and other technologies to standardize resource description.

RDF describes resources and their relationships through *triples* of the form (*s*, *p*, *o*), where: *s* is the *subject* of the triple, which is an RDF URI reference or a blank node; *p* is the *predicate* or *property* of the triple, which is an RDF URI reference, and it specifies how *s* and *o* are related; and *o* is the *object*, which is an RDF URI reference, a literal or a blank node. A triple (*s*, *p*, *o*) may also be denoted as "<*s*><*p*><*o*>".

A *tripleset* is just a set of triples. In this paper, we will use *dataset* and *tripleset* interchangeably.

RDF Schema is a semantic extension of RDF to cover the description of classes and properties of resources. OWL (W3C OWL Working Group, 2012) in turn extends RDF Schema to allow richer descriptions of schemas and ontologies, including cardinality and other features.

RDF Schema and OWL define the following predicates that we will use in the rest of the thesis:

- `rdfs:subClassOf` indicates that the subject of the triple defines a subclass of the class defined by the object of the triple
- `owl:sameAs` indicates that the subject denotes the same concept as the object
- `rdfs:seeAlso` indicates that the subject is generically related to the object

- `owl:equivalentClass` indicates that both the subject and the object are classes and denote the same concept
- `rdf:type` indicates that the subject is an instance of the object

For example, the triple

```
<dbpedia:Sweden>  <rdf:type>  <dbpedia:Country>.
```

indicates that the resource `dbpedia:Sweden` is an instance of the class `dbpedia:Country`.

Triplesets are typically available on the Web as SPARQL endpoints (Prud'hommeaux et al., 2012) or as file dumps (large files containing all the data from a tripleset, or small files containing only the relevant data for a defined term). A third option is through URL dereferencing, which means that the resource contains descriptive data about itself so it is possible to discover more data simply by reading the resource content.

SPARQL is a query language and a protocol. As a query language, it works similarly to SQL: it is possible to query databases over a specific resource, join resources and limit data to a determined parameter. As a protocol, it defines the query interface (HTTP), how requests should be made (POST or GET) and how the data should be returned (via a standard XML). Thus, an agent can perform queries on a dataset and acquire knowledge to create new queries and so on.

On March 2013, the SPARQL 1.1 specification (Garlik et al., 2013) was published with added SQL-like grouping functions. It allowed us, for instance, to count the number of triples with a given property. For example, this extension allows counting the number of instances of a class *C* in a tripleset, that is, to count how many triples in the tripleset have `rdf:type` as property and *C* as object.

VoID (Alexander et al., 2009) is an ontology used to define metadata about triplesets. A VoID document is a good source of information about a tripleset, such as the classes and properties it uses, the size of the tripleset, and etc.

Let *d* be a tripleset and *V* be a set of VoID metadata descriptions. The classes and properties used in *d* can be extracted from tripleset partitions defined by the properties `void:classPartition` and `void:propertyPartition` that occur in *V*. *Class partitions* describe sets of triples related to subjects of a particular class. *Property partitions* describe sets of triples that use a particular predicate. These partitions are described by the properties `void:class` and

`void:property` respectively. The set of vocabulary terms used in *d* can be generated by the union of all values of the properties `void:class` and `void:property`. In some cases, the VoID description of a tripleset does not define partitions, but specifies a list of namespaces of the vocabularies used by the tripleset with the `void:vocabulary` predicate. One can enrich the set of vocabulary terms used in *d* with such a list.

Finally, the Web site Datahub.io stores descriptions of some of the triplesets in the Linking Open Data Cloud (LOD Cloud). The Web site is built using an open source tool called Ckan[4], which has mechanisms to programmatically discover triplesets published in the LOD Cloud.

2.2. Evaluation of tools

Throughout this thesis, we will compare our approach with other state-of-art approach for semantic web crawling. To have a fair comparison, we adopted three different metrics that are widely used in the literature for this kind of comparison.

1. Precision – identifies the number of relevant resources when compared to all resources found by the crawler (in our case). It is defined as follows:

$$Precision = \frac{|RLT|}{|RT|}$$

where:

- RLT = number of relevant resources retrieved
- RT = number of resources retrieved

Precision shows how the tool handles the values it found in order to retrieve only the relevant resources.

2. Recall – compares the relevant resources found to the whole set of relevant resources available. It is defined as follows:

$$Recall = \frac{|RLT|}{|RL|}$$

where:

- RLT = number of relevant resources retrieved

[4] http://ckan.org/

- *RL* = number of relevant resources

Recall shows how good the tool is to discover all relevant resources.

3. F-Measure (F_1 score) (Manning et al., 2008) – tests the accuracy of a tool. It considers both *precision* and *recall* is defined, as follows:

$$\mathrm{F1} = \frac{precision * recall}{precision + recall}$$

In other words, the F_1 score can be interpreted as the weighted average of *precision* and recall.

3
Related Work

3.1. Traditional Web Crawlers

Web crawling is a common task on the Web (Baeza-Yates and Berthier, 1999). Since the beginning of the World Wide Web (WWW), unstructured information distributed in dozens of servers are being indexed in order to facilitate the search. Web Crawlers are tools intended to discover information on Web sites and can be classified as: exhaustive and topical (or focused) crawlers. Exhaustive crawlers meet the needs of the general population of Web users. On the other hand, topical crawlers are activated in response to a particular information need as they expect an initial input to start (Srinivasan et al., 2005).

In the specific area of topical crawlers, we highlight the following contributions next. FishSearch (De Bra et al., 1994) is one of the first proposals for focused crawlers. The system works as follows: the user provides a starting URL and a match condition, which could be a set of keywords or a regular expression. The crawler starts to search for pages and saves those in which the content matches the specified condition. In order to decide which page the tool will crawl next, the system uses a priority queue of unvisited URLs with the priority being defined by a simplified scoring module. SharkSearch (Herscovici et al., 1998) is an evolution of the FishSearch crawler that uses more contextual data to calculate a cosine-based relevance score to define the priority.

De Assis (De Assis et al., 2009) presents a genre-aware approach to focused crawling. In addition to searching for any relevant information related to the initial input, the proposed algorithm focuses on documents of a specified genre. Furthermore, the paper states that Web focused crawlers normally use some kinds of contextual information to estimate the benefit of a URL:

- Link Context: is the information available *nearby* the link to the URL on the Web page.
- Ancestor Pages: uses the content of the ancestors' pages to determine the relevance of the URL.

- Web Graph: uses a Web sub-graph around the page associated with the URL to decide whether to follow the URL.

3.2. Linked Data Crawlers

The crawlers proposed in this thesis are similar to topical crawlers as they expect an initial input to start their crawling. However, while Web crawlers read unstructured information, Linked Data crawlers – such as ours – work with structured information, in the form of RDF triples. Furthermore, the crawlers proposed in this thesis use RDF metadata information (such as `rdfs:subClassOf` and `owl:sameAs`) as contextual information, which is far simpler than what topical crawlers consider as contextual information.

We now describe some of the most popular Linked Data crawlers available. Ding et al. (2005) present a tool created by Swoogle to discover new triplesets. The authors describe a way of ranking Web objects in three granularities: Web documents (Web pages with embedded RDF data), terms and RDF Graphs (triplesets). Each of these objects has a specific ranking strategy.

Hartig (Hartig et al., 2009) introduces a Semantic Web client that considers all the Linked Data as a single SPARQL Endpoint that will fill the resultset based on its crawling. The crawling method works following RDF links from one resource to another.

LDSpider (Isele et al., 2010) is another example of a Linked Data crawler. Similarly to the crawlers proposed in this thesis, LDSpider starts with a set of URIs as a guide to parse Linked Data.

Fionda et al. (2012) present a language, called NAUTILOD, which allows browsing through nodes of a Linked Data graph. They introduced a tool, called *swget* (semantic web get), which evaluates expressions of the language. An example would be: “find me information about Rome, starting with its definition in DBpedia and looking in DBpedia, Freebase and the New York Times databases”.

```
swget <dbp:Rome>
  (<owl:sameAs>)* -saveGraph-domains {dbpedia.org,
   rdf.freebase.com, data.nytimes.com}
```

The Linked Data crawlers just described have some degree of relationship with the proposed crawlers, though none has exactly the same goals. The crawlers

proposed in this thesis focus on finding metadata that are useful to design new triplesets. Furthermore, rather than just dereferencing URIs, they adopts SPARQL crawling queries to improve recall, as explained in Section 4.5.

The crawlers proposed in this thesis depend on SPARQL endpoints, which are by nature a valuable and sometimes sporadic resource. Berners-Lee (Berners-Lee, 2006), in one of the first papers introducing the Linked Data, wrote: "*to make the data be effectively linked, someone who only has the URI of something must be able to find their way to the SPARQL endpoint.*" Besides, many triplesets do not have a valid SPARQL endpoint (see section 7.6). Three recent papers tried to address this issue using different approaches.

Linked Data Fragments (Verborgh et al., 2014) argue that, instead of having a generic query interface that accepts any sort of queries and, therefore, suffers from performance issues, the publisher should create low-cost queryable data that use client CPU in addition to server processing. The solution, although feasible, depends on the implementation of new servers and clients able to process this new type of information.

The LOD Laundromat (Wouter et al., 2014), on the other hand, provides a more viable approach: instead of expecting endpoints and clients to change, it provides a large set of triplesets from the Linked Data in a third endpoint. This endpoint, in addiction, undergoes a cleaning process in order to facilitate automatic processing by other tools.

Contrasting with the LOD Laundromat, Roomba (Assaf et al., 2015) is a tool that proposes to extract, validate, correct and generate linked dataset profiles. The created profile may help evaluate a tripleset for a decision process and avoid spamming triplesets.

3.3. Tripleset Recommendation

We now comment on how the proposed crawlers relate to recommender tools for Linked Data.

Some generic recommender tools use keywords as input. Nikolov et al. (2011, 2012) use keywords to search for relevant resources, using the label property of the resources. Indeed, a label is a property used to provide a human-

readable version of the name of the resource[5]. A label value may be inaccurate, in another language or simply be a synonymous of the desired word. There is no compromise with the schema and its relationships. Therefore, the risk of finding an irrelevant resource is high.

Martínez-Romero et al. (2010) propose an approach for the automatic recommendation of ontologies based on three points: (1) how well the ontology matches the set of keywords; (2) the semantic density of the ontology found; and (3) the popularity of the tripleset on the Web 2.0. They also match a set of keywords to resource label values, in a complex process.

The crawlers proposed in this thesis may be used as a component of a recommender tool, such as those just described, to locate: (1) appropriate ontologies during the triplification of a database; (2) triplesets to interlink with a given tripleset. We stress that the crawlers were not designed to be a full recommender tool, but rather to be a component of one such system.

[5] http://www.w3.org/TR/rdf-schema/#ch_label

4
A Linked Data Crawling Strategy

4.1.
Introduction

This chapter first introduces the crawling approach adopted in the remaining chapters.

Then, it describes the proposed metadata crawling strategy. Section 4.3 covers how the strategy simulates a breadth-first search for new terms and triplesets, whereas Section 4.5 discusses the use of SPARQL *crawling queries* and *URI dereferencing* to find new terms and triplesets.

The chapter concludes with a brief discussion on how to use VoID to extract more information about triplesets.

4.2.
Examples of the Proposed Crawling Strategy

4.2.1. A schematic example

Figure 2 and Figure 3 compare the traditional focused crawling strategy and the strategy proposed in this thesis. Figure 1 explains the symbols presented in both figures, where the directions of the arrows indicate in which side the resource was found.

A traditional crawler uses an element already visited to locate other elements directly related to it. For example, in Figure 2, the crawler is able to move from element *A* to elements *B* and *C,* which will be crawled at a second level.

The strategy presented in this thesis (Figure 3) uses the traditional strategy and adds other ways to discover elements, with the help of SPARQL queries, as detailed in Section 4.5. In the example, from element *A*, the crawler finds elements *B* and *C*, as in previous example, but it is also able to find elements *F*, *G*, *H* and *I* using SPARQL queries. Such queries are applied over all triplesets available in datahub.io and some common ontologies.

Figure 3 also illustrates another specific scenario: when a third ontology describes the relationship between two elements – note that *M* and *G* are related by an RDF triple stored elsewhere (represented by the dashed box in Figure 3).

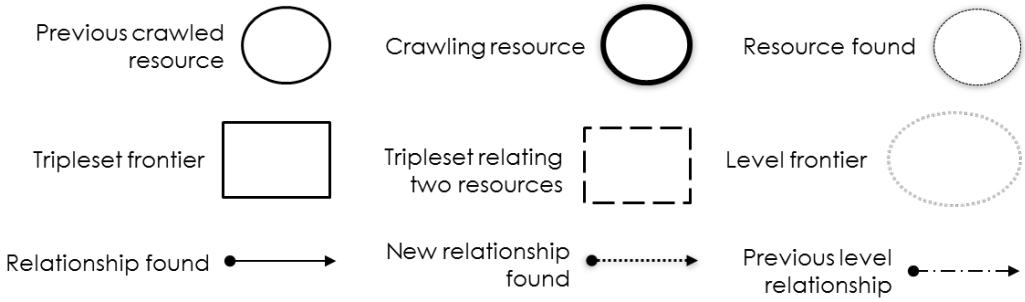

Figure 1. Legend of both strategies

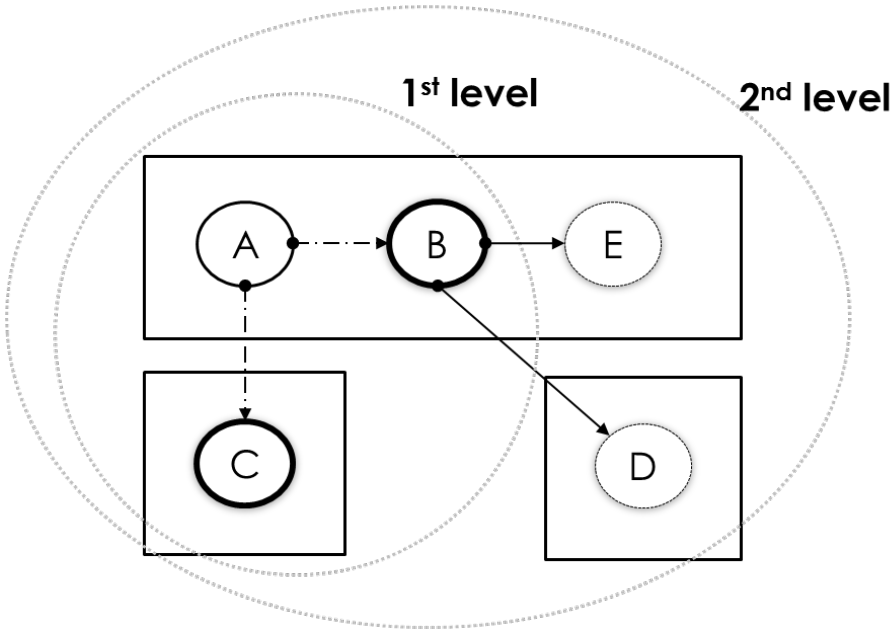

Figure 2. Traditional focused crawling strategy

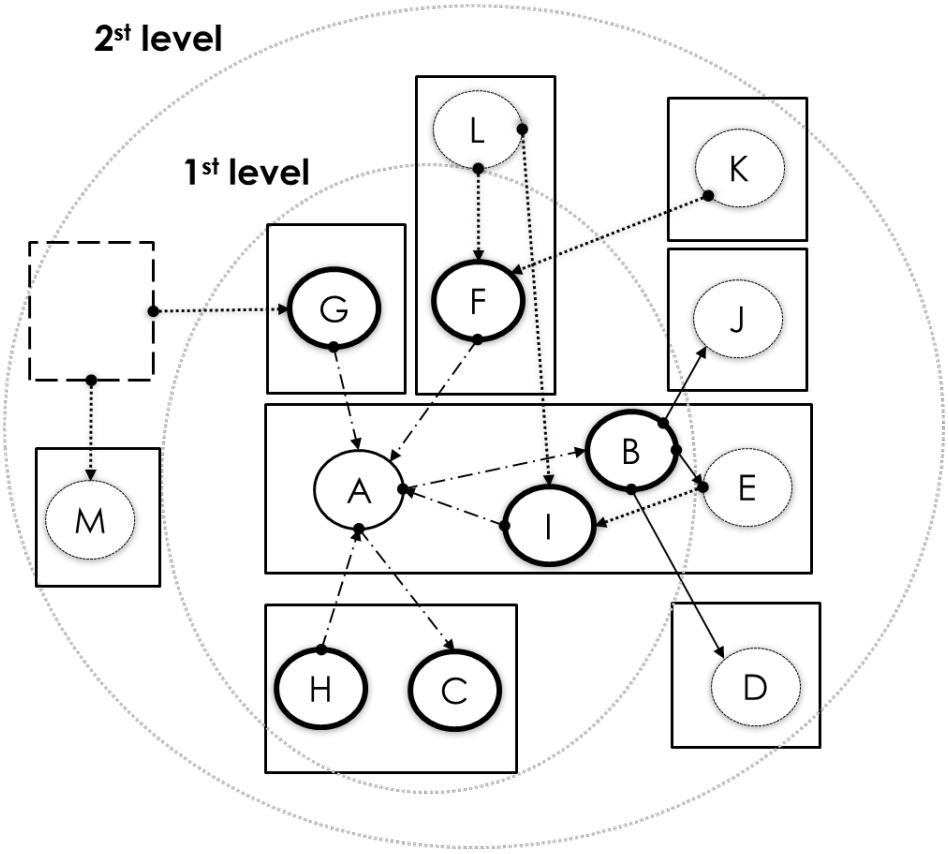

Figure 3. Thesis strategy on focused crawling

4.2.2. A Concrete Use Case

Consider a user who wants to publish as Linked Data a relational database *d* storing music data (artists, records, songs, etc.). A metadata crawler designed along the strategy proposed in this thesis will help the user publish *d* as follows.

First, the user has to define an initial set *T* of terms to describe the application domain of *d*. Suppose that he selects just the term `dbpedia:Music`, taken from DBpedia.

The user will then invoke the metadata crawler, passing *T* as input. The crawler will query the Datahub.io catalogue of Linked Data triplesets to crawl triplesets searching for new terms that are directly or transitively related to `dbpedia:Music`. The crawler focuses on finding new terms that are defined as subclasses of the class `dbpedia:Music`, or that are related to `dbpedia:Music` by

`owl:sameAs` or `rdfs:seeAlso` properties. The crawler will also count the number of instances of the classes found.

The crawler will return: (1) the list of terms found, indicating their provenance – how the terms are direct or transitively related to `dbpedia:Music` and in which triplesets they were found; (2) for each class found, an estimation of the number of instances in each tripleset visited; and (3) a list relating the VoID data of each tripleset with each one of the terms found.

The user may take advantage of the results that the crawler returned in two ways. They may manually analyze the data and decide: (1) which of the probed ontologies found they will adopt to triplify the relational database; and (2) to which triplesets the crawler located they will link the tripleset they are constructing. Alternatively, they may submit the results of the crawler to separate tools that will automatically recommend the ontologies to be adopted in the triplification process, as well as the triplesets to be used in the linkage process (Leme et al., 2013; Lopes et al., 2013).

For example, suppose that the crawler finds two subclasses, `opencyc:Love_Song` and `opencyc:Hit_Song`, of `wordnet:synset-music-noun-1` in the ontology `opencyc:Music`. Suppose also that the crawler finds large numbers of instances of these subclasses in two triplesets, `musicBrains` and `bbcMusic`. The user might then decide that `opencyc:Music` is a good ontology to adopt in the triplification process, and that `musicBrains` and `bbcMusic` are good triplesets to use in the linkage process.

4.3. Breadth-First Search for New Terms

We assume that the metadata crawling process receives as input:

- A set of catalogues that identify SPARQL endpoints and RDF dumps, possibly augmented with manually informed triplesets. The end-points and dumps used are collectively called *input triplesets,* or simply *triplesets* in what follows.
- A set of terms *T*, called the *initial crawling terms*. Such terms are typically selected from generic ontologies, such as WordNet, DBpedia, and Schema.org , albeit this is not a requirement for the crawling process.

The crawling strategy dictates that the crawler must simulate a breadth-first search for new terms. *Level 0* contains the initial set of terms *T*. The set of terms of each new level is computed from those of the previous level with the help of the crawling queries and URI dereferencing, as described in Section 0, except for `rdf:type`, which is used only to count the number of instances found. Figure 4 illustrates the level-based crawling strategy.

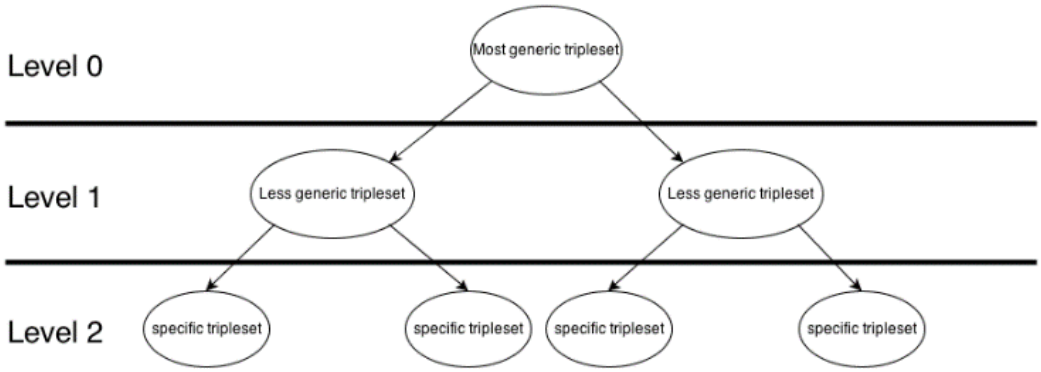

Figure 4. Crawling levels.

The *crawling frontier* is the set of terms found that have not yet been processed. To avoid circular references, we mark the terms that have already been processed.

For each new term found, the crawler creates a list that indicates the provenance of the term: how the term is directly or transitively related to an initial term and in which tripleset(s) it was found. That is, the crawler identifies the sequence of relationships it traversed to reach a term, such as in the following example:

```
wordnet:synset-music-noun-1 -> owl:sameAs -> OpenCyc:Music ->
rdfs:subClassOf -> OpenCyc:LoveSong ->
instance -> 500 instances.
```

Figure 5. Example of provenance.

4.4. Parameters

Since the number of terms may grow exponentially from one level to the next, we prune the search by limiting:

- The number of levels of the breadth-first search the tool will crawl to. If a small number of levels is defined, the tool may be unable to reach relevant resources (recall). But, if a larger number of levels is defined, the tool may start to lose its precision. Chapter 7 shows a complete evaluation of the parameters.
- The maximum number of terms probed. This parameter restricts the number of terms that will be crawled. In some experiments, we noticed that a higher level of terms probed can increase the time of the experiment, without gaining relevant information in the same proportion.
- The maximum number of terms probed for each term in the crawling frontier. Similarly to the previous parameter, it restricts the number of crawling resources that were found when crawling a previous resource. The idea of this argument is to balance the number of resources crawled throughout the tool.
- The maximum number of terms probed in each tripleset, for each term in the crawling frontier. It is similar to the previous parameter, but it also restricts the dataset where the resources will be searched. In some experiments, we noticed that a single dataset retrieved over a thousand relevant resources. Since the dataset was the first to be processed, any other resource found by the crawler will be ignored by the previous parameter. This parameter avoids this case by creating a new restriction.

4.5. Crawling Queries and URI Dereferencing

The crawling queries find new terms that are related to the terms obtained in the previous level through the following *crawling properties:* `rdfs:subClassOf`, `owl:sameAs` and `rdfs:seeAlso`. Hence, these queries are respectively called *subclass*, *sameAs* and *seeAlso* queries.

Figure 6 shows one of the templates of the *crawling queries* that obtain terms related to a known term *t* through the crawling property *p*.

```
SELECT distinct ?item
WHERE { ?item p <t> }
```

Figure 6. Template query to obtain a subset of the crawling results.

For the properties `owl:sameAs` and `rdfs:seeAlso`, the crawler must also use the template query of Figure 7. For each term *t* to be crawled, it inverts the role of *t*, as shown in Figure 7, when the predicate *p* is `owl:sameAs` and `rdfs:seeAlso`, since these predicates are reflexive, and it is reasonable that the description of the term itself will be explained in that order. However, the crawler must not invert the role of *t* when the predicate *p* is `rdfs:subClassOf`, since this predicate is not reflexive.

```
SELECT distinct ?item
WHERE { <t> p ?item }
```

Figure 7. Template of the inverted SPARQL query.

Consider the crawling property `rdfs:subClassOf`. Suppose that *C* and *C'* are classes defined in triplesets *S* and *S'*, respectively, and assume that *C'* is declared as a subclass of *C* through a triple of the form

(C', `rdfs:subClassOf`*, C)*

Triples such as this are more likely to be included in the tripleset where the more specific class *C'* is defined than in the tripleset where the more generic class *C* is defined. Hence, after finding a class *C*, the crawler must search for subclasses of *C* in all triplesets it has access using the template of Figure 6.

Another case occurs when the relationship between *C* and *C'* is defined in a third schema *S''*. Similar to the previous example, we need a subclass query over *S''* to discover that the relationship between *C* and *C'*. *S''* is obtained by dereferencing the URI of *C'*. In most cases, the returned tripleset is the complete ontology where *C'* is defined, while in some other cases only a fragment of the ontology where *C'* is defined is returned.

A special type of crawling query is obtained by replacing *p* in Figure 6 with `rdf:type`. However, in this case, only the overall number of instances found and the total number of instances for each tripleset are retrieved and stored in the result set of the crawling process.

Finally, the crawling process also uses URI dereferencing as follows: for each term that will be crawled, the tool accesses its URI RDF content. The content

will them be interpreted as new, small, tripleset to which the same SPARQL queries are applied, as shown in Figure 6 and Figure 7.

4.6. Using VoID to Extract more Information about Triplesets

The crawler will eventually collect a large number of terms and count the number of instances of a reasonable number of classes, declared in many triplesets. These data can be used to extract more metadata about a tripleset by parsing its VoID description, as follows.

For each tripleset *t* in the catalogues the crawler uses, if *t* has a VoID description *V*, the crawler retrieves all objects *o* from triples of the form (*s*, `void:class`, *o*) declared in *V*. The resources retrieved are compared to all resources the crawler already located. Each new resource found is saved and returned as part of the final output of the entire crawling operation, with an indication that it is also related to tripleset *t* through a VoID description.

Although the crawling process has limiting parameters to avoid time-consuming tasks, the processing of VoID descriptions is simple enough and, therefore, not subjected to limitations.

4.7. Summary

At a very high level of abstraction, , the strategy presented in this thesis can be described as the following pseudo-code (more detailed descriptions will be available in the following chapters).

```
CRAWLER(maxLevels, Terms, Catalogues; Provenance)

Parameters: maxLevels    - maximum number of levels of the breadth-first search
input:      Terms        - a set of input terms
            Catalogues   - a list of catalogues of triplesets
output:     Provenance   - a provenance list for the terms in Q
begin
    var currentLevel = 0
    var nextLevelTerms = [ ]
    while currentLevel < maxLevels
        foreach term T on Terms:
            foreach catalogue C on Catalogues
                Query C to count the number of instances of T
                Query C to search for relationship properties T and add to nextLevelTerms
                Add all provenance of the queries above to Provenance
            end
        end
        Terms = nextLevelTerms
    end
    return Provenance
end
```

5
A Proof of Concept of the Metadata Crawling Strategy

5.1.
Introduction

To evaluate the concept described in Chapter 4, we created a simple tool, in which its pseudo-code is shown in Annex A. The tool was implemented in *Java* using the framework *Apache Jena*[6] to resolve Linked Data resources.

The tool was only created to evaluate the metadata crawling strategy and how it performed, when compared with other Linked Data crawlers. Chapters 6 and 7 describe implementation alternatives that address the performance issues observed in the proof-of-concept implementation.

In the rest of this thesis, we refer to this first implementation simply as the *proof of concept crawler*.

5.2.
Experiments

5.2.1.Organization of the Experiments

We evaluated the *proof of concept crawler* over triplesets described in *Datahub.io*. The tool was able to recover 317 triplesets with SPARQL endpoints. However, despite this number, it could run queries on just over half of the triplesets due to errors in the query parser, or simply because the servers were not available.

To execute the tests, we separated three set of terms related to the music and publication application domains. To create the initial crawling terms, we used three generic ontologies, WordNet, DBpedia, and Schema.org, as well ontologies specific to each domain, as described in Section 5.2.2.

WordNet is a lexical database that presents different meanings for the same word. For example, the term `wordnet:synset-music-noun-1` means “an artistic

[6] https://jena.apache.org/

form of auditory communication incorporating instrumental or vocal tones in a structured and continuous manner"[7]. In addition, the term `wordnet:synset-music-noun-2`[8] is defined as "any agreeable (pleasing and harmonious) sounds; "he fell asleep to the music of the wind chimes"".

DBpedia is the triplified version of the Wikipedia database. The triplification process is automatically accomplished and the current English version has already 2.5 million classified items.

Schema.org is the most recent ontology of all three. It focuses on HTML semantics and was created by Google, Bing, and Yahoo. Therefore, Schema.org is now used by many triplesets[9]. Schema.org is also developing other ways to increase the search results by creating a mapping with other ontologies, such as DBpedia and WordNet.

We elected these three ontologies as the most generic ones. All three have a collection of terms that covers numerous domains and could be used together to determine an initial set that represents the user's intentions. Of course, if a user has good knowledge about a domain, they can adopt more specific ontologies to determine the initial crawling terms. In the examples that follow, we use the abbreviations shown in Table 1.

Table 1. Namespace abbreviation.

Abbreviation	Namespace
`akt`	http://www.aktors.org/ontology/portal#
`bbcMusic`	http://linkeddata.uriburner.com/about/id/entity/http/www.bbc.co.uk/music/
`dbpedia`	http://dbpedia.org/resource/
`dbtune`	http://dbtune.org/
`freebase`	http://freebase.com/
`freedesktop`	http://freedesktop.org/standards/xesam/1.0/core#
`lastfm`	http://linkeddata.uriburner.com/about/id/entity/http/www.last.fm/music/
`mo`	http://purl.org/ontology/mo/
`musicBrainz`	http://dbtune.org/musicbrainz/
`nerdeurocom`	http://nerd.eurecom.fr/ontology#
`opencyc`	http://sw.opencyc.org/2009/04/07/concept/en/
`schema`	http://schema.org/
`twitter`	http://linkeddata.uriburner.com/about/id/entity/http/twitter.com/
`umbel`	http://umbel.org/

[7] http://goo.gl/TIKswe

[8] http://goo.gl/TIKswe

[9] http://schema.rdfs.org/mappings.html

wordnet	http://wordnet.rkbexplorer.com/id/
yago	http://yago-knowledge/resource/

The experiments used the following parameters (see section 4.3), arbitrarily defined. Section 7 presents a complete evaluation of these parameters:

- Number of levels: 2
- Maximum number of terms probed: 40
- Maximum number of terms probed for each term in the crawling frontier: 20
- Maximum number of terms probed in each tripleset, for each term in the crawling frontier: 10

The experiments ran over Azure Virtual Machines[10], using an A4 instance (8 cores, 14GB of RAM).

5.2.2. Results

Music Domain

We chose music as the first domain to evaluate the basic crawler and elected three ontologies, DBpedia, WordNet and the Music Ontology[11], to select the initial crawling terms. The Music Ontology is a widely accepted ontology that describes music, albums, artists, shows, and some specific subjects.

The initial crawling terms were:

```
mo:MusicArtist
mo:MusicalWork
mo:Composition
dbpedia:MusicalWork
dbpedia:Song
dbpedia:Album
dbpedia:MusicalArtist
dbpedia:Single
wordnet:synset-music-noun-1
```

In what follows, we will first comment on the results obtained at Level 1, for each initial term. Then, we will proceed to discuss how the new terms obtained in Level 1 were processed at Level 2.

[10] https://azure.microsoft.com

[11] http://musicontology.com/

Table 2 (a) shows the results of Level 1 for `mo:MusicalArtist`. At Level 2, for each of the terms `mo:MusicGroup` and `mo:SoloMusicArtist`, the basic crawler obtained similar results: nearly 2,000 resources were found in the triplesets `bbcMusic` and `musicBrainz:data`, which are large databases about the music domain; and the *seeAlso* query pointed to an artist, `lastfm:Hadas`. As *seeAlso* provides additional data about the subject, we speculate that the result the basic crawler returned represents a mistake made by the database creator.

Table 2(b) shows the results of Level 1 for `mo:MusicalWork`. Note that the basic crawler found a variety of instances from multiple databases, mainly on universities. At Level 2, when processing `mo:Movement`, the basic crawler found a *seeAlso* reference to `lastfm:Altmodisch`.

At Level 1, when processing `mo:Composition`, the basic crawler found 13 instances, but no related terms.

Table 2(c) shows the results of Level 1 for the first DBpedia term, `dbpedia:MusicalWork`. The basic crawler found 5 subclasses from DBpedia and more than 20,000 subclasses from the `yago` tripleset. This unusual result is due to the segmentation used by `yago`. For example, there are subclasses that segment records by artist, by historical period, and even by both. The first three terms, `dbpedia:Album`, `dbpedia:Song` and `dbpedia:Single`, will be analyzed in the next paragraphs since they are also in the initial set of terms.

Table 2. Related terms.

Query type	Description
(a) Related terms for `mo:MusicArtist`	
subclass	`mo:MusicGroup` `mo:SoloMusicArtist`
instance	103,541 instances, mostly from `lastfm`
(b) Related terms for `mo:MusicalWork`	
subclass	`mo:Movement`
instance	16,833 instances found in multiple databases like `dbtune` and academic music databases
(c) Related terms for `dbpedia:MusicalWork`	

subclass	`dbpedia:Album`
	`dbpedia:Song`
	`dbpedia:Single`
	`dbpedia:Opera`
	`dbpedia:ArtistDiscography`
	`and 21,413 classes from yago`

sameAs	dbpedia:MusicGenre umbel:MusicalComposition
seeAlso	lastfm:Syfin lastfm:Kipling lastfm:Pandemic lastfm:Ardcore lastfm:Lysis lastfm:Freakhouse lastfm:Saramah lastfm:Akouphen lastfm:Freakazoids lastfm:Cyrenic lastfm:Phender twitter:Ariadne_bullet
instance	145,656 instances
(d) Related terms for dbpedia:Song	
Own URL	dbpedia:EurovisionSongContestEntry
sameAs	schema:MusicRecording
subclass	dbpedia:EurovisionSongContestEntry
seeAlso	lastfm:Apogee lastfm:Brahman lastfm:Anatakikou lastfm:Sakerock lastfm:8otto lastfm:Cro-Magon lastfm:Ladz Plus 7 lastfm resources in Japanese
instance	10,987 instances from multiple language versions of dbpedia, lastfm and others
(e) Related terms for dbpedia:Album	
Own URL	freebase:en.Album opencyc:Album
subclass	nerdeurocom:Album and 17,222 subclasses, mostly from yago
sameAs	schema:MusicAlbum freebase:en.Album dbpedia:Sophomore_Album and some dbpedia:Album classes from other Wikipedia languages
instance	100,090 instances from multiple language versions of dbpedia and others
(f) Related terms for dbpedia:MusicalArtist	
seeAlso	lastfm:Krackhead
sameAs	dbpedia:Musician umbel:MusicalPerformer
subclass	dbpedia:Instrumentalist dbpedia:BackScene and 2,178 subclasses from yago
instance	49,973 instances from multiple language versions of dbpedia
(g) Related terms for dbpedia:Single	
seeAlso	last.fm:Toxin last.fm:Dethrone

	last.fm:Burdeos last.fm:Sylence twitter:joint_popo last.fm:Toximia last.fm:Alcoholokaust last.fm:Electromatic last.fm:Mighty+Atomics
subclass	3,414 subclasses, the majority from yago
instance	44,623 instances

At Level 2, the processing of dbpedia:Opera returned no results and the processing of dbpedia:ArtistDiscography returned 3,423 instances, but no new term. The processing of umbel:MusicalComposition returned 1,809 instances, and dbpedia:MusicGenre retrieved 7,808 new instances.

Table 2(d) shows the results of Level 1 for dbpedia:Song. The basic crawler found the most diversified results in terms of query types and query results. It was able to identify resources in different languages (such as Portuguese and Greek), which was only possible because it focused on metadata. Crawlers that use text fields (Nikolov and d'Aquin, 2011) can only retrieve data in the same language as that of initial terms.

At Level 2, when processing dbpedia:EurovisionSongContestEntry, the basic crawler obtained three subclasses from yago, a *sameAs* relationship with schema:MusicRecording and found the same result of dbpedia:Song for the *seeAlso* property. The other resource probed on the Level 2 was schema:MusicRecording, which returned no instances or new crawling terms.

Table 2(e) shows the results of Level 1 for dbpedia:Album. The processing of this term also produced an interesting result. The *sameAs* query found a small number of unique relationships, but found some dbpedia:Album in other languages. One may highlight the opencyc:Album class, for which the basic crawler was able to find 245 instances.

Table 2(f) shows the results of Level 1 for dbpedia:MusicalArtist. The processing of this term exhibited results similar to those obtained by processing dbpedia:Album, in terms of quantity of subclasses. Therefore, it was possible to recover results in multiple languages.

On Level 2, when processing dbpedia:Musician, the basic crawler found over 163 *sameAs* terms, the majority of them pointing to DBpedia in other languages (even in non-latin alphabets). On the other hand, the *seeAlso* query

found over 50 terms, but none of them seems related to the subject. When processing `umbel:MusicalPerformer`, the basic crawler retrieved one subclass, `umbel:Rapper`, and over 6,755 instances from a variety of triplesets.

Table 2(g) shows the results of Level 1 for `dbpedia:Single`. As for other resources from DBpedia, the basic crawler was able to find a large number of subclasses from `yago` tripleset. In addition, it found more than 40,000 instances from different triplesets in many languages.

The last term probed in Level 1 was `wordnet:synset-music-noun-1`. The basic crawler found a *sameAs* relationship with an analogue term from another publisher: `wordnet:synset-music-noun-1`. At Level 2, the basic crawler found a new *sameAs* relationship to `opencyc:Music`.

Finally, we remark that, when we selected the terms to evaluate, we expected to find relationships between DBpedia and Music Ontology, which did not happen. In addition, we found much better results using terms from DBpedia than from the Music Ontology, which is specific to the domain in question. The definition of links between the Music Ontology and DBpedia could increase the popularity of the former. For example, if the term `mo:MusicArtist` were related to the term `dbpedia:MusicalArtist`, crawlers such as ours would be able to identify the relationship. Also, matching or recommendation tools would benefit from such relationship.

Publications domain

For the second domain, we focused on two ontologies, Schema.org and Aktors[12], which are commonly used by publications databases. We selected the following terms:

```
schema:TechArticle
schema:ScholarlyArticle
akt:Article-Reference
akt:Article-In-A-Composite-Publication
akt:Book, akt:Thesis-Reference akt:Periodical-Publication
akt:Lecturer-In-Academia
akt:Journal
```

[12] http://www.aktors.org

The results were quite simple. While the queries based on Schema.org practically returned no results, queries on Aktors returned enough instances, but with no complex structure. A quick analysis showed that almost all triplesets were obtained from popular publications databases (such as DBLP, IEEE, and ACM) by the same provider (RKBExplorer), which used the Aktors ontology. In addition, the Aktors ontology is not linked to other ontologies, which lead to an almost independent cluster in the Linked Data cloud.

The VoID processing, as discussed in Section 4.6, was not able to find any new information. In fact, in a more detailed analysis, it was clear that VoID seems to be a neglected feature. From the initial 317 triplesets, only 102 had the VoID description stored in Datahub.io, and only 8 had any triple with the property `void:class` (which were not related to our test domains).

Processing times

Table 3 shows the processing time for each experiment. In general, the time spent to process each term was directly related to the number of terms found (some exceptions apply due to bandwidth issues).

Table 3 shows that the minimum time was 14 minutes, when no new terms were found, but the maximum time depended on the number of new terms in the crawling frontier, and how the network (and the endpoints) responded.

Finally, we observe that the processing time can be optimized, provided that: (1) the endpoints queries have lower latency; (2) the available bandwidth is stable across the entire test; (4) cache features are used; (3) queries are optimized to reduce the number of requests.

Table 3. Performance evaluation.

Term	Proc. time (minutes)
Music domain	
`mo:MusicArtist`	70
`mo:MusicalWork`	28
`mo:Composition`	14
`dbpedia:MusicalWork`	183
`dbpedia:Song`	163
`dbpedia:Album`	173
`dbpedia:MusicalArtist`	167
`dbpedia:Single`	186
`wordnet:synset-music-noun-1`	24

Publications domain	
schema:TechArticle	29
schema:ScholarlyArticle	47
akt:Article-Reference	14
akt:Article-In-A-Composite-Publication	28
akt:Book	14
akt:Thesis-Reference	14
akt:Periodical-Publication	28
akt:Lecturer-In-Academia	14
akt:Journal	14

5.2.3. A comparison with SWGET

We opted for a direct comparison between the *proof-of-concept crawler* and *swget* for three reasons. First, there is no benchmark available to test Linked Data crawlers such as ours, and it is nearly impossible to manually produce one such (extensive) benchmark. Second, *swget* is the most recent crawler available online. Third, it was fairly simple to setup an experiment for *swget* similar to that described in Section 5.2.2 for the music domain. We decided to restrict the evaluation to the music domain, since the publication domain does not have relationships between ontologies that can lead for new triplesets (see section 5.2.2).

Briefly, the experiment with *swget* was executed as follows. Based on the examples available at the *swget* website, we created the following template to run queries (where *t'* is the term to be probed and *q'* the current crawling property):

```
t' -p <q'> <2-2>
```

The above query means "given a term *t'*, find all resources related to it using the predicate *q*' expanding two levels recursively.

Then, we collected all terms *swget* found from the same initial terms of the music domain used in Section 5.2, specifying which crawled property *swget* should follow. Table 4 shows the number of terms *swget* found, for each term and crawling property.

Table 4. Number of terms found using *swget*.

Term	subclass	sameAs	seeAlso	type
mo:MusicArtist	4	0	0	3
mo:MusicalWork	7	0	0	3
mo:Composition	0	0	0	3

dbpedia:MusicalWork	16	1	0	3
dbpedia:Song	6	1	0	3
dbpedia:Album	6	1	0	3
dbpedia:MusicalArtist	9	1	0	3
dbpedia:Single	6	1	0	3

Based on the experiments with *swget* and the basic crawler, we compiled the list of terms shown in Table 5. We excluded the terms retrieved from `yago` to avoid unbalancing the experiment in favor of the basic crawler. Then, we manually inspected the terms and marked, in Table 5, those that pertain to the music domain, and those that *swget* and the basic crawler found.

Table 5. Comparison between SWGET and the Basic Crawler

	Terms retrieved by *swget* or crawler	Manual Validation	Swget	Crawler
(Terms retrieved by swget)				
dbpedia:MusicalWork		-	-	-
1	dbpedia:Song	Y	Y	Y
2	dbpedia:Single	Y	Y	Y
3	dbpedia:Album	Y	Y	Y
4	dbpedia:Work	N	Y	N
5	dbpedia:ArtistDiscography	Y	Y	Y
6	dbpedia:Opera	Y	Y	Y
7	dbpedia:EurovisionSongContestEntry	Y	Y	Y
8	owl:Thing	N	Y	N
9	dbpedia:Software	N	Y	N
10	dbpedia:RadioProgram	N	Y	N
11	dbpedia:Cartoon	N	Y	N
12	dbpedia:TelevisionSeason	N	Y	N
13	dbpedia:Film	N	Y	N
14	dbpedia:Website	N	Y	N
15	dbpedia:CollectionOfValuables	N	Y	N
16	dbpedia:WrittenWork	N	Y	N
17	dbpedia:Musical	Y	Y	N
18	dbpedia:Artwork	N	Y	N
19	dbpedia:LineOfFashion	N	Y	N
20	dbpedia:TelevisionShow	N	Y	N
21	dbpedia:TelevisionEpisode	N	Y	N
22	dbpedia:Song	Y	Y	Y
23	dbpedia:Single	Y	Y	Y
dbpedia:MusicalArtist		-	-	-
24	dbpedia:Artist	N	Y	N
25	schema:MusicGroup	Y	Y	N
26	dbpedia:Sculptor	N	Y	N
27	dbpedia:Painter	N	Y	N
28	dbpedia:Actor	N	Y	N

29	dbpedia:ComicsCreator	N	Y	N
30	dbpedia:Comedian	N	Y	N
31	dbpedia:FashionDesigner	N	Y	N
32	dbpedia:Writer	N	Y	N
33	dbpedia:Person	N	Y	N
dbpedia:Song		-	-	-
	(No new term retrieved swget)			
dbpedia:Album		-	-	-
	(No new term retrieved swget)			
dbpedia:Single		-	-	-
	(No new term retrieved swget)			
mo:MusicArtist		-	-	-
34	mo:SoloMusicArtist	Y	Y	Y
35	foaf:Agent	Y	Y	N
36	mo:MusicGroup	Y	Y	Y
37	foaf:Person	Y	Y	N
38	foaf:Organization	Y	Y	N
mo:MusicalWork		-	-	-
39	mo:Movement	Y	Y	Y
40	frbr:Work	N	Y	N
41	frbr:ScholarlyWork	N	Y	N
42	frbr:ClassicalWork	N	Y	N
43	frbr:LegalWork	N	Y	N
44	frbr:LiteraryWork	N	Y	N
45	frbr:Endeavour	N	Y	N
46	wordnet:Work~2	N	Y	N
mo:Composition		-	-	-
	(No term retrieved)			
(Terms retrieved only by crawler)				
47	umbel:MusicalComposition	Y	N	Y
48	schema:MusicRecording	Y	N	Y
49	freebase:en.Album	Y	N	Y
50	opencyc:Music	Y	N	Y
51	opencyc:Album	Y	N	Y
52	nerdeurocom:Album	Y	N	Y
53	schema:MusicAlbum	Y	N	Y
54	dbpedia:Sophomore_Album	Y	N	Y
55	dbpedia:Musician	Y	N	Y
56	umbel:MusicalPerformer	Y	N	Y
57	umbel:Rapper	Y	N	N
58	dbpedia:Instrumentalist	Y	N	Y
59	dbpedia:BackScene	N	N	Y
60	dbpedia:MusicGenre	Y	N	Y
61	freebase:en.Album	Y	N	Y
	36 items from lastfm	Y	N	Y
	2 items from twitter	N	N	Y

The results detailed in Table 5 can be summarized by computing the precision and recall obtained by *swget* and the basic crawler for the list of terms as follows:

- Column Headers / Values:
  - Manual Validation:
    - Y = term relevant for the Music domain
    - N = term not relevant for the Music domain
  - Retrieved by *swget* and retrieved by Basic Crawler:
    - Y = term retrieved by *swget* or Basic Crawler
    - N = term not retrieved by *swget* or Basic Crawler
- Terms retrieved by swget or Basic Crawler:
  - Retrieved terms: 99
  - Relevant terms that were retrieved (identified by "Y" in column "Manual Validation"): 66
- Terms retrieved by swget:
  - Retrieved terms: 46
  - Relevant terms that were retrieved (identified by rows with the pattern (Y,Y,-)): 16
  - Precision = 16 / 46 = 0.35
  - Recall = 16 / 66 = 0.24
- Terms retrieved by the Basic Crawler:
  - Retrieved terms: 63
  - Relevant terms that were retrieved (identified by rows with the pattern (Y,-,Y)): 60
  - Precision = 60 /63 = 0.95
  - Recall = 60/66 = 0.91

Briefly, both tools archive the following metric's value:

```
swget:              precision = 35%   recall = 24%
basic crawler:      precision = 95%   recall = 91%
```

These results should be interpreted as follows. *Swget* achieved a much lower precision since it finds more generic and more specific terms at the same time, while the basic crawler only searches for the more specific terms. This feature creates undesirable results for the purposes of focusing on an application domain. For example, using `rdfs:subClassOf` as predicate and `dbpedia:MusicalWork` as object, *swget* returned `dbpedia:Work`, a superclass at the first level. At the next level, *swget* then found resources such as `dbpedia:Software` and `dbpedia:Film`, each of them subclasses of `dbpedia:Work`, but unrelated to the Music domain.

The basic crawler achieved a better recall in part since, given two classes defined in different triplesets, it was able to uncover relationships between the

classes described in a third tripleset. Indeed, *swget* processed `umbel:MusicalPerformer` using properties `rdfs:subClassOf` and `owl:sameAs`. Our expectation was that it would be able to find the class `dbpedia:MusicalWork`, as the basic crawler div, which did not happen. A quick analysis showed that the relationship between both classes was not described in any of the original triplesets, but in a third tripleset, `http://linkeddata.uriburner.com/`.

This behavior should not be regarded as defect of *swget* though, but a consequence of working with a general-purpose crawler, rather than a metadata focused crawler, such as ours.

To conclude, the simple *proof of concept crawler* was able to outperform the state-of-art crawling tool for the semantic web. However, our tool had performance issues that must to be addressed.

5.3. Lessons Learned

In this section, we highlight the main lessons learned from the first implementation of a crawler that follows the strategy proposed in Chapter 4. We first enumerate some aspects that may influence the crawling results, such as the settings of the parameters and the availability of sufficient information about the crawled triplesets.

Parameter setting. Since, in the basic crawler, the set of terms of each new level is computed from that of the previous level, the number of terms may grow exponentially. We defined some parameters to prune the search. Hence, the user must adequately set such parameters to obtain results in reasonable time, without losing essential information.

Choosing the initial crawling terms. In the music domain experiments, we started with terms from three different triplesets, DBpedia, WordNet, and Music Ontology, the first two being more generic than the last one. It seems that the resources defined in the Music Ontology are not interlinked (directly or indirectly) with the more popular triplesets. This limitation is related to the fact that some triplesets do not adequately follow the Linked Data principles, in the sense that

they do not interlink their resources with resources defined in other relevant triplesets.

Ontologies describing the domain of interest. The basic crawler proved to return more useful data when there are relationships among the metadata. In the experiments using the publications domain, the basic crawler returned a simplified result, because all triplesets related to the initial crawling terms used the same ontology to describe their resources. In general, the larger the number of triplesets in the domain, the more useful the results of the basic crawler will be.

VoID description. The VoID processing seems to be an adequate solution to a faster access to tripleset information. Despite the VoID expressivity, most triplesets used in our experiments had a simplistic VoID description available. Hence, the basic crawler hardly found new data using the VoID descriptions.

6
CrawlerLD – An Optimized Implementation of the Metadata Crawling Strategy

6.1.
Introduction

The implementation presented in Chapter 5 showed that the proposed metadata crawling strategy was effective. Indeed, probing resources by level and using crawling queries to discover new resources returned better results, when compared to a state-of-art crawler.

We may, however, enumerate points that need to be improved and points that need to be corrected:

Organization – It is possible to divide the previous implementation into four steps: dereferencing, property crawling, instance counter, and VoID analysis. All these steps are distributed throughout the implementation, without a clear separation.

Expanding techniques – The previous implementation had four clearly defined processors, but the entire crawling mechanism might be used for other purposes beyond those identified so far. Thus, it is desired to provide a *plug-and-play* mechanism to allow other developers to create their own crawling processor. Conversely, these custom processors might be useful for metadata crawling.

Time performance – Although we extracted good results, the time spent waiting for a response is infeasible. In our latest experiments, the best result time was 14 minutes (nothing was found) and the worst, 3 hours. We have to improve the processing time to make the crawling strategy feasible.

To address these issues, we created a second implementation, engineered as a framework, in which its pseudo-code is listed in Annex B. In the rest of this

thesis, we refer to this second implementation as the *optimized crawler or CrawlerLD*. We continue to refer to the implementation described in Chapter 5 as the *basic crawler*.

The optimized crawler also receives as input a set of initial crawling terms *T*. Given *T*, the optimized crawler uses a list *C* of *processors*, described in Section 6.3, in successive levels (see Section 4.3), to extract new terms from the triplesets listed in the catalogues. Each processor annotates the provenance of its crawled data and returns a list of terms to be crawled in the next level, after filtering, based on parameters specified by the user (see also Section 4.3). Besides an architecture based on processors, the optimized crawler incorporates improvements to the crawling queries, outlined in Section.6.2.

As described in Section 6.4, to evaluate the optimized crawler, we reapplied the experiments detailed in Section 5.2.

6.2. Improvements to the crawling queries

The optimized crawler incorporates several changes to the crawling queries to reduce the number of request and improve the precision of the results. The changes will be better illustrated in Section 6.3.

Replacement of `rdfs:seeAlso` ***by*** `owl:equivalentClass`.

In the evaluation of the basic crawler, we discovered that the `rdfs:seeAlso` property would decrease the precision of the crawling task. We therefore replaced it by `owl:equivalentClass`, which is mostly used to map ontologies. For example, the schema.org ontology has RDF mapping files that use `owl:equivalentClass` to create relationships to other consolidated ontologies (such as DBPedia).

Property query changed.

The basic crawler searched each property individually, increasing the number of queries it had to execute. The optimized crawler uses a unique query (see Figure 8) that combines all properties. In fact, we reduced the number of queries by 5 (three properties and two that are reflexive), for each crawling resource.

Instance counter changed.

The basic crawler asked for all instances of a resource that are stored in the endpoint of each tripleset, this query have two disadvantages: (1) it spends too much bandwidth; (2) it creates overhead to the endpoint and also to the tool. To address this problem, we changed the query to use a grouping function (see section 6.3 - Instance Counter processor).

6.3. A Processor Architecture

CrawlerLD, the optimized crawler, includes three processors, described further in this section.

Dereference processor

The first processor is responsible for extracting information of the resource itself. As described in Section 4.5, it tries to find new resources using the properties `owl:sameAs`, `owl:equivalentClass`, and `rdfs:subclassOf`. For each such property, the processor applies a SPARQL query to extract new information. The following template illustrates how each query works, where *p* is one of the above properties and *t* is the crawling term itself; the values assigned to the variable `?item` are resources to be crawled in a next level.

```
SELECT distinct ?item
 WHERE {<t> p ?item}
```

Given that `owl:sameAs` and `owl:equivalentClass` are reflexive, the processor also applies SPARQL queries generated by a new code template, with the subject and object inverted:

```
SELECT distinct ?item
 WHERE {?item p <t>}
```

Property processor

This processor is responsible for crawling other datasets. It uses a special SPARQL query, which runs over each dataset discovered in DataHub and manually added as described Section 4.3. The motivation is to extract information that is not directly related to the resources already processed. Given the crawling term *t* that will be processed by the crawler and a dataset *d* that uses *t* to describe a

fraction of its data. While a conventional crawling algorithm is not able to find *d* since *t* does not have any reference to *d*. This crawler, on the other hand, traverses all datasets available and is able to find the relationship between *d* and *t.*

The processor uses the SPARQL template shown in Figure 8, where *t* is the resourced being crawled. Note that this SPARQL template essentially combines all templates shown in Figure 6 and Figure 7, which avoids the overhead of calling the SPARQL endpoint several times.

```
SELECT distinct ?property ?item
WHERE {
        { ?item owl:sameAs <t>.}
  UNION { <t> owl:sameAs ?item.}
  UNION { ?item owl:equivalentClass <t>.}
  UNION { <t> owl:equivalentClass ?item.}
  UNION { ?item rdfs:subClassOf <t>.}
  ?item ?property <t>. }
```

Figure 8. Property query.

Note that, for each term *t* to be crawled, the template inverts the role of *t* (for the details, see lines 7 and 9 of the code in Annex B), when the predicate is `owl:sameAs` and `owl:equivalentClass`, since these predicates are reflexive. However, the crawler does not invert the role of *t*, when the predicate is `rdfs:subClassOf`, since this predicate is not reflexive.

For example, in the specific case of the crawling property `rdfs:subClassOf`, suppose that *C* and *C'* are classes defined in triplesets *S* and *S'*, respectively, and assume that *C'* is declared as a subclass of *C* through a triple of the form

(*C'*, `rdfs:subClassOf`, *C*)

Triples such as this are more likely to be included in the tripleset where the more specific class *C'* is defined than in the tripleset where the more generic class *C* is defined. Hence, after finding a class *C*, the crawler has to search for subclasses of *C* in all triplesets it has access to, using the template above.

Another case occurs when the relationship between *C* and *C'* is defined in a third ontology *S''*. Similar to the previous example, we need a subclass query over *S''* to discover that *C'* is a subclass of *C*. *S''* is obtained by dereferencing the URI

of *C'*. In most cases, the returned tripleset is the complete ontology where *C'* is defined, while in some other cases only a fragment of the ontology where *C'* is defined is returned.

Instance Counter processor

The last processor extracts information about the quantity of instances available in each dataset for each crawling term. It runs queries over all datasets, using the same principle as the property processor. To reduce the bandwidth, the processor uses grouping functions (Figure 9) to query datasets.

```
SELECT distinct (count(?instance) AS ?item)
WHERE { ?instance rdf:type <%s> . }
```

Figure 9. Applying grouping function to calculate the number of instances.

Unfortunately, grouping functions are only available in SPARQL 1.1 (Garlik et al., 2013) and above. Therefore, the processor also crawls the remaining datasets using an alternative query (Figure 10), which spends more bandwidth.

```
SELECT distinct ?item
WHERE { ?item rdf:type <%s> . }
```

Figure 10. Alternative instance counter query.

6.4. Experiments

6.4.1. Organization of the Experiments

To evaluate the optimized crawler, we re-executed the experiments described in Section 5.2. In addition, we added more triplesets extracted from DataHub.io and reached 1,042 datasets that had a SPARQL endpoint or a RDF Dump. We also added the mapping ontologies that relate Schema.org to other popular ontologies[13]. However, over half of these datasets are duplicated, and the optimized crawler was able to run queries on just over 35% of triplesets due to errors in the query parser, or simply because the servers were not available. In

[13] http://schema.rdfs.org/mappings.html

addition, over 6 months separated both experiments (January, 2014 and July, 2014), which could also affect the comparison.

To summarize, even after six months and having a larger list of triplesets, we were not able to find relationships between DBpedia and Music Ontology. We again found much better results using terms from DBpedia than Music Ontology. Also, when comparing the optimized version against the basic crawler described in Chapter 5, we discovered that the optimized crawler found less resources than the previous one. Indeed, we found that some triplesets were not available at the time of our experiment. The results for the publications domain were quite similar to those for the basic crawler, reported in Section 5.2.2.

The experiments ran over Azure Virtual Machines[14], using an A7 instance (8 cores, 56GB of RAM).

The rest of this section may be skipped on a first reading since it shows results very similar to those of Section 5.2. The reader may go directly to the topic *Processing times* in Section 6.4.2, which shows significant differences between the optimized and the basic implementations.

6.4.2. Results

The experiments involved the same domains of the first crawler, Music and Publications, and the same parameters, arbitrarily defined. Section 7 presents a complete evaluation of the parameters:

- Number of levels: 2.
- Maximum number of terms probed: 40.
- Maximum number of terms probed for each term in the crawling frontier: 20.
- Maximum number of terms probed in each tripleset, for each term in the crawling frontier: 10.

Music Domain

The first domain used to evaluate the crawler was Music and three ontologies were elected to select the initial crawling terms, DBpedia, WordNet and Music

[14] https://azure.microsoft.com

Ontology[15]. The Music Ontology is a widely accepted ontology that describes music, albums, artists, shows and some specific subjects.

The initial crawling terms were:

`mo:MusicArtist`
`mo:MusicalWork`
`mo:Composition`
`dbpedia:Album`
`dbpedia:MusicalArtist`
`dbpedia:Single`
`dbpedia:MusicalWork`
`dbpedia:Song`
`wordnet:synset-music-noun-1`

Next, we comment on the results obtained in Level 1, for each initial term. Then, we discuss how the new terms obtained in Level 1 were processed in Level 2.

Table 6(a) shows the results of Level 1 for `mo:MusicalArtist`. On Level 2, for each of the terms `mo:MusicGroup` and `mo:SoloMusicArtist`, the crawler obtained different results: while `mo:MusicGroup` recovered over 1.5 million instances over three datasets, `mo:SoloMusicArtist` did not find any new result.

Table 6(b) shows the results of Level 1 for `mo:MusicalWork`. Note that the crawler found a variety of instances from multiple databases. On Level 2, when processing `mo:Movement`, the crawler did not find any new instance or class.

Table 6(c) shows the results of Level 1 for the first DBpedia term, `dbpedia:MusicalWork`. The crawler found 5 subclasses from DBpedia and over a million instances in 13 datasets, with 8 being DBpedia in different languages (such as French, Japanese, Greek, and others), which was only possible because it focused on metadata. Crawlers that use text fields (Nikolov et a., 2011) can only retrieve data in the same language as that of initial terms.

The first three terms, `dbpedia:Album`, `dbpedia:Song`, and `dbpedia:Single`, will be analyzed in the next paragraphs since they are also in the initial set of terms.

On Level 2, the processing of `dbpedia:Opera` returned no results and the processing of `dbpedia:ArtistDiscography` returned 48,784 instances, but no new term.

[15] http://musicontology.com/

Table 6. **Related terms**

Query type	Description
(a) Related terms for `mo:MusicArtist`	
subclass	`mo:MusicGroup, mo:SoloMusicArtist`
instance	2,647,957 instances from over four datasets
(b) Related terms for *`mo:MusicalWork`*	
subclass	`mo:Movement`
instance	1,166,365 instances found in multiple databases
(c) Related terms for `dbpedia:MusicalWork`	
subclass	`dbpedia:Album, dbpedia:Song, dbpedia:Single, dbpedia:Opera, dbpedia:ArtistDiscography`
instance	939,480 instances from 13 datasets
(d) Related terms for `dbpedia:Song`	
equivalentclass	`schema:MusicRecording`
subclass	`dbpedia:EurovisionSongContestEntry`
instance	35,702 instances from 9 datasets
(e) Related terms for `dbpedia:Album`	
equivalentclass	`schema:MusicAlbum`
instance	871,348 instances from 13 datasets
(f) Related terms for `dbpedia:MusicalArtist`	
instance	424,152 instances from 19 datasets
(g) Related terms for `dbpedia:Single`	
instance	305,041 instances from 10 datasets

Table 6(d) shows the results of Level 1 for `dbpedia:Song`. The crawler was able to find a relationship with other generic dataset (Schema.org) and also found a variety of resources from DBpedia in different languages.

On Level 2, when processing `dbpedia:EurovisionSongContestEntry`, the crawler found 7,807 instances from 7 datasets. The other resource probed on the Level 2 was `schema:MusicRecording`, which returned 38,464 instances and no new crawling terms.

Table 6(e) shows the results of Level 1 for `dbpedia:Album`. The processing of this term also found `schema:MusicAlbum` and a large number of instances. On Level 2, the tool was able to find 662,409 instances of `schema:MusicAlbum`, but no new resource.

Table 6(f) shows the results of Level 1 for `dbpedia:MusicalArtist`. The tool was not able to find any new related resource, but it found a large number of datasets that have instances of this class.

Table 6(g) shows the results of Level 1 for `dbpedia:Single`. The tool found more than 300 thousand instances from triplesets in many languages.

The last term probed in Level 1 was `wordnet:synset-music-noun-1`. The crawler found a *sameAs* relationship with an analogue term from another publisher: `http://www.w3.org/2006/03/wn/wn20/instances/synset-music-noun-1`.

Publications domain

For the second domain, we focused on two ontologies, Schema.org and Aktors[16], which are commonly used by publications databases. We selected the following terms:

```
schema:TechArticle
schema:ScholarlyArticle
akt:Article-Reference
akt:Article-In-A-Composite-Publication
akt:Book, akt:Thesis-Reference akt:Periodical-Publication
akt:Lecturer-In-Academia
akt:Journal
```

The results for the publications domain were quite similar to those for the basic crawler, reported in Section 5.2.2. Both ontologies (Schema.org and Aktors) returned a small number of instances, but with no complex structure. A quick analysis showed that almost all triplesets were obtained from popular publications databases (such as DBLP, IEEE and ACM) by the same provider (RKBExplorer), which uses the Aktors ontology. In addition, the Aktors ontology is not linked to other ontologies, which lead to an almost independent cluster in the Linked Data cloud.

Processing times

Table 7 shows the processing time for each experiment with the optimized crawler. In general, the time spent to process each term was directly related to the number of terms found (some exceptions apply due to bandwidth issues). The

[16] http://www.aktors.org

experiment was performed on a virtual machine hosted by Microsoft Azure[17] with 56GB and two AMD Opteron™ 4171 processors.

Table 7 shows that the minimum time was 4 minutes, when no new terms were found, but the maximum time depended on the number of new terms in the crawling frontier, and how the network (and the endpoints) responded.

Finally, we observe that the processing time can be optimized, provided that: (1) the endpoints queries have lower latency; (2) the available bandwidth is stable across the entire test; (3) cache features are used.

Table 7. **Performance evaluation**

Term	Proc. time (minutes)
Music domain	
mo:MusicArtist	11
mo:MusicalWork	8
mo:Composition	4
dbpedia:MusicalWork	22
dbpedia:Song	11
dbpedia:Album	8
dbpedia:MusicalArtist	4
dbpedia:Single	4
wordnet:synset-music-noun-1	11
Publications domain	
schema:TechArticle	4
schema:ScholarlyArticle	4
akt:Article-Reference	4
akt:Article-In-A-Composite-Publication	8
akt:Book	5
akt:Thesis-Reference	5
akt:Periodical-Publication	4
akt:Lecturer-In-Academia	5
akt:Journal	4

6.4.3. A new comparison with SWGET

In this section, we compare the optimized crawler again with swget for the music domain, but in a different scenario, as explained in Section 6.4.1. To mitigate the tripleset availability problem, we executed swget simultaneously with

[17] http://azure.microsoft.com/

the optimized implementation. Table 8 shows the number of new terms *swget* found for each initial term and crawling property.

Table 8. **Number of terms found using *swget.***

Initial Term	Crawling Property			
	subclass	***sameAs***	***equivalentclass***	***type***
mo:MusicArtist	6	0	0	0
mo:MusicalWork	8	0	0	0
dbpedia:MusicalWork	21	0	0	0
dbpedia:Song	7	0	1	0
dbpedia:Album	6	0	1	0
dbpedia:MusicalArtist	10	0	0	0
dbpedia:Single	6	0	0	0

Based on the experiments with *swget*, we compiled a list of terms shown in Table 9. Then, we manually inspected the terms and marked those that pertain to the Music domain and those that *swget* and this crawler found.

Table 9. Comparison between SWGET and CrawlerLD.

	Terms retrieved by *swget* or CRAWLER-LD	Manual Validation	Swget	Crawler
(Terms retrieved by *swget*)				
dbpedia:MusicalWork		-	-	-
1	dbpedia:Song	Y	Y	Y
2	dbpedia:Single	Y	Y	Y
3	dbpedia:Album	Y	Y	Y
4	dbpedia:Work	N	Y	N
5	dbpedia:ArtistDiscography	Y	Y	Y
6	dbpedia:Opera	Y	Y	Y
7	dbpedia:EurovisionSongContestEntry	Y	Y	Y
8	owl:Thing	N	Y	N
9	dbpedia:Software	N	Y	N
10	dbpedia:RadioProgram	N	Y	N
11	dbpedia:Cartoon	N	Y	N
12	dbpedia:TelevisionSeason	N	Y	N
13	dbpedia:Film	N	Y	N
14	dbpedia:Website	N	Y	N

15	dbpedia:CollectionOfValuables	N	Y	N
16	dbpedia:WrittenWork	N	Y	N
17	dbpedia:Musical	Y	Y	N
18	dbpedia:Artwork	N	Y	N
19	dbpedia:LineOfFashion	N	Y	N
20	dbpedia:TelevisionShow	N	Y	N
21	dbpedia:TelevisionEpisode	N	Y	N
dbpedia:MusicalArtist		-	-	-
22	dbpedia:Artist	N	Y	N
23	schema:MusicGroup	Y	Y	N
24	dbpedia:Sculptor	N	Y	N
25	dbpedia:Painter	N	Y	N
26	dbpedia:Actor	N	Y	N
27	dbpedia:ComicsCreator	N	Y	N
28	dbpedia:Comedian	N	Y	N
29	dbpedia:FashionDesigner	N	Y	N
30	dbpedia:Writer	N	Y	N
31	dbpedia:Person	N	Y	N
dbpedia:Song		-	-	-
32	schema:MusicRecording	Y	Y	Y
33	dbpedia:MusicalWork	Y	Y	N
dbpedia:Album		-	-	-
34	schema:MusicAlbum	Y	Y	Y
dbpedia:Single		-	-	-
	(No new term retrieved swget)			
mo:MusicArtist		-	-	-
35	mo:SoloMusicArtist	Y	Y	Y
36	foaf:Agent	N	Y	N
37	mo:MusicGroup	Y	Y	Y
38	foaf:Person	N	Y	N
39	foaf:Organization	N	Y	N
40	foaf:Group	N	Y	N
mo:MusicalWork		-	-	-
41	mo:Movement	Y	Y	Y
42	frbr:Work	N	Y	N
43	frbr:ScholarlyWork	N	Y	N
44	frbr:ClassicalWork	N	Y	N
45	frbr:LegalWork	N	Y	N
46	frbr:LiteraryWork	N	Y	N

47	frbr:Endeavour	N	Y	N
48	wordnet:Work~2	N	Y	N
mo:Composition				
	(No terms retrieved)			
(Terms retrieved only by CRAWLER-LD)				
	(No terms retrieved)			

The results can be summarized by computing the precision, recall and balanced F-measure (F_1) obtained by *swget* and the optimized implementation for the list of terms as follows:

- Column Headers / Values:
  - Manual Validation:
    - Y = term relevant for the Music domain
    - N = term not relevant for the Music domain
  - Retrieved by *swget* and retrieved by CRAWLER-LD:
    - Y = term retrieved by *swget* or CRAWLER-LD
    - N = term not retrieved by *swget* or CRAWLER-LD
- Terms retrieved by swget or CRAWLER-LD:
  - Retrieved terms: 48
  - Relevant terms that were retrieved (identified by "Y" in column "Manual Validation"): 14
- Terms retrieved by swget:
  - Retrieved terms: 48
  - Relevant terms that were retrieved (identified by rows with the pattern (Y,Y,-)): 14
  - Precision = 14 / 48 = 0.2917
  - Recall = 14 / 14 = 1.0
  - F_1-Measure = 2 * ((0.2917*1.0) / (0.2917+1.0)) = 0.4516
- Terms retrieved by CRAWLERLD:
  - Retrieved terms: 11
  - Relevant terms that were retrieved (identified by rows with the pattern (Y,-,Y)): 11
  - Precision = 11 / 11 = 1.0
  - Recall = 11 / 14 = 0.7857
  - F_1-Measure = 2 * ((1.0*0.7857) / (1.0+0.7857)) = 0.8800

Briefly, both tools archive the following metric value:

```
swget:
    Precision = 29.17%   Recall = 100%     F1 = 45.16%
Optimized crawler:
    Precision = 100%     Recall = 78.57%   F1 = 88.00%
```

Recall from Section 5.2.3 that, when we compared the basic crawler with swget for the music domain, we obtained the following results:

```
swget:          Precision = 35%  Recall = 24%
Basic crawler:  Precision = 95%  Recall = 91%
```

The new results (for swget versus the optimized crawler) are therefore somewhat similar to the old ones (for swget versus the basic crawler). Comparing precision, swget fell over 5%, while the optimized crawler increased to 100%. On the other hand, the recall of swget jumped from 24% to 100%, and the optimized crawler decreased nearly 12,5%. However, we observe that some triplesets present in the first experiment were not available at the time of this second experiment. Indeed, some triplesets that also returned relevant resources in Chapter 5 (and swget was unable to discover) were offline. This accounts for the increase in the recall of swget.

Analyzing the overall quality of the crawlers using F-measure, our crawler outperformed *swget*, obtaining an F1 result almost twice as large as that of *swget*. Thus, in this experiment, our crawler was able to find a better balance between recall and precision values than *swget*.

To conclude, we noticed a decrease in performance when comparing this implementation with *proof-of-concept crawler* tool in section 5, although this implementation also outperform SWGET. We can enumerate the difference between each version of our approach as follows:

1 – Some datasets were not available at the time of the evaluation. Unfortunately, these datasets were responsible for many links found in our previous evaluation;

2 – Our decision to remove the *seeAlso* property as a crawling property. The property decreased our precision in the *proof-of-concept* approach and we found out that precision was was important than recall.

3 – Some unexpected behaviors were identified from performance issues like a high memory footprint.

6.5. Lessons Learned

In this section, we highlight the main lessons learned from the development of the optimized crawler and the results of our experiments.

Reducing the number of request. Our crawling strategy demands a high number of requests to each tripleset. Hence, creating ways to reduce this number would

improve performance. Our approach, primarily implemented on the *property processor,* combines all queries into a single one, using the *UNION* clause and processing the result set locally.

Tripleset availability. Even with a larger set of triplesets, the optimized crawler was not able to find some resources found by the basic crawler. The output of a crawler based on SPARQL queries is indeed volatile and depends on tripleset availability at the moment of the execution of the crawler. A solution to reduce the unpredictability of a result is to use previous results as a foundation for the new one.

Distribution. With the solutions proposed in this chapter, we reduced the number of queries we made to each tripleset. Although, the processing times of the optimized crawler reduced significantly, to reach better results we will need to adopt a distributed, scalable architecture.

Architectural problems. The optimized crawler presented in this chapter, although faster than the basic crawler, has many performance problems that need to be addressed. Due to the memory consumption problems, we were only able to execute crawling tasks that started with just one initial term. Also, any resource that returned a large number of new resources created difficulties when probed. In fact, we were only able to run the experiments described in Section 6.4, using a machine that had 54GB of main memory.

7
DIST-CrawlerLD – An Actor Model-based Implementation of the Metadata Strategy

7.1. Introduction

Earlier implementations of the crawling strategy had problems that need to be addressed in order to increase usability and decrease resource consumption:

1. High memory footprint – previous experiments have shown that the amount of available memory may be insufficient, depending on the specified task parameters. One of our experiments topped 50GB of heap memory and stopped working at the third crawling level. This issue showed us that our tool is not scalable.
2. Time consuming – for each crawling term, CrawlerLD need to make several queries to distributed triplesets. Each term would take 15 to 30 minutes to process. If we process a hundred terms we will need 50 hours in the worst case, which is infeasible.
3. Scalability – the crawler is "locked" into a single machine. In complex use case scenarios, this limitation will be an issue. The tool need to be able to process using more than one machine.
4. Lack of a user interface – every process in CrawlerLD (optimized implementation) is done through command lines. Even the crawler's result has to be analyzed through files and unfriendly tools.

This chapter describes how we addressed the first three topics by adopting the actor model; the solution for the fourth topic also benefited from this approach.

The new implementation, called *DIST-CrawlerLD*, should be viewed as a re-engineering of CrawlerLD, the optimized implementation of Chapter 6. Hence, both tools have the same inputs and use the notion of processor to implement the crawling strategy. Whenever necessary, we will continue to refer to the implementation described in Chapter 5 as the *basic implementation*.

7.2. The actor model

This section briefly reviews concepts from the Actor Model, adopted in the implementation of the metadata crawling tool described in this chapter.

The reactive manifesto[18] is a document elaborated by developers to make better scalable software. Briefly, a *reactive software* must have the following characteristics:

- Responsiveness – to respond in a timely manner, if at all possible.
- Resilience – To stay responsive in the face of a failure.
- Elasticity – To stay responsive under varying workload.
- Be message driven – To rely on asynchronous message-passing and to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provide the means to delegate errors as messages.

First presented by Hewitt (Hewitt et al., 1973), the actor model is one of the models that address all characteristics of the reactive manifesto. It is described as a model of concurrent computation, in which an actor is an isolated computing unit. An actor has its own state and only one thread executing at a time. This computing unit receives messages, makes decisions, and can create other actors or send new messages to address its objective. The actor model also encourages the separation of the software into small pieces of code that are engineered to receive and send messages to other pieces of code.

Figure 11 shows an example of the actor model. The execution flow starts with *Actor 0* sending a message to *Actor 3*. *Actor 3* evaluates the content of the message and decides if it needs to send new requests to *Actors 1* and *2*. While both actors are processing their messages, *Actor 1* sends a new request to *Actor 2*, which is not immediately processed, since *Actor 2* is still processing the message from *Actor 3*. After processing both messages, *Actor 2* sends a reply to *Actor 0*. Note that *Actor 0* will not be blocked during such processing: once it sends the message to *Actor 3*, it can address other messages while waiting for the message from *Actor 2*.

[18] http://www.reactivemanifesto.org/

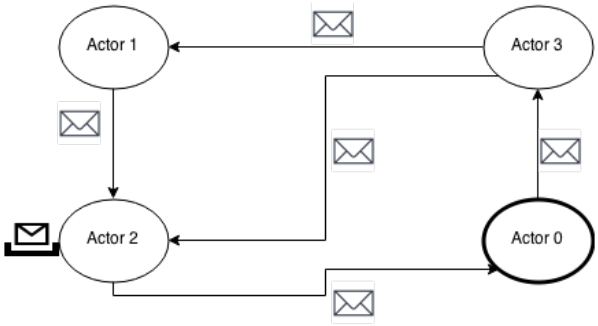

Figure 11. An example of the Actor Model.

The model was chosen to be applied on the crawler for a number of reasons:

1. It uses a responsive design: the code is not blocked by another thread, since it does not have to wait for another task to finish. CrawlerLD suffers from blocking thread issues since, it has to wait a processor to complete its task before sending a result.
2. It addresses the module by adopting the actors model, which is similar to our concept of *processors* (section 6.3).
3. The message exchange between actors can be automatically queued without any effort from our part;
4. It facilitates creating a distributed version of the tool.

7.3. An Actor Model-based Architecture

7.3.1. Software Architecture

To make DIST-CrawlerLD easier to develop and deploy, we modularized the tool as follows (see Figure 12):

- **CrawlerLD.core** – the core system, embeds all logic created to crawl the LOD cloud. It can be used as an API for other tools and allows any third-party developer to create new processors. This module heavily uses the **UtilitiesSemanticWeb** library to crawl the LOD cloud.
- **CrawlerLD.gui** – integrates the REST service (for a microservice architecture) and the graphical user interface. It uses the **CrawlerLD.core** as its backend.
- **CrawlerLD.command-line** – allows the user to crawl the LOD cloud using a command prompt interface.

- **CrawlerLD.distribution** – is designed to be deployed over remote machines to enable distributed computing. It encapsulates all dependencies needed to run the tool in a distributed environment.
- **UtilitiesSemanticWeb (USW)** – a library created to facilitate access to the LOD cloud. It uses Apache Jena and is able to execute several SPARQL requests to remote and local endpoints and retrieve new datasets from DataHub.

The next sections describe some of the implementation details of the **CrawlerLD.core**, the **UtilitiesSemanticWeb** and the **CrawlerLD.gui** modules.

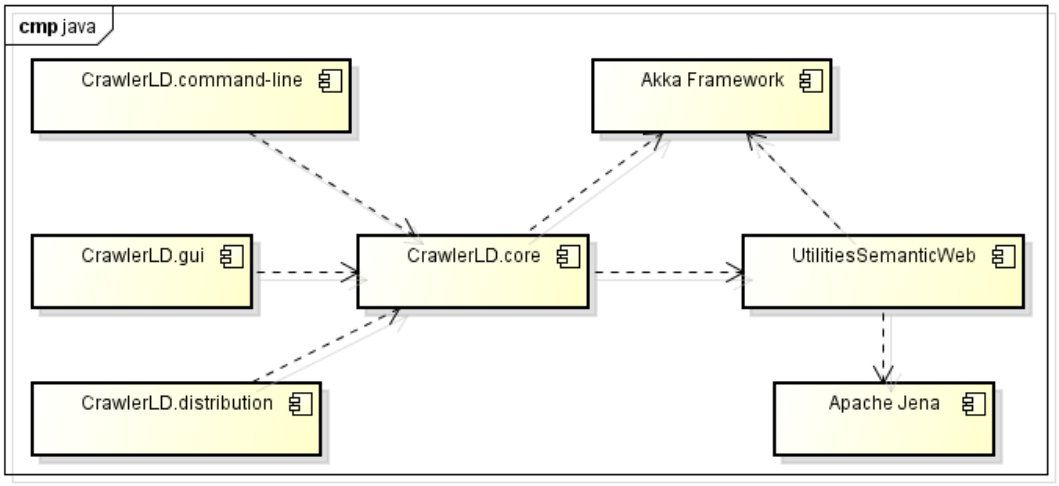

Figure 12. CrawlerLD modules and dependencies

7.3.2. Tripleset availability test

A large number of triplesets are available at the Linked Data cloud and can be accessed using the datahub.io catalog. This catalog, however, is not updated frequently. In special, it may fail to report when a resource is no longer available.

In our experience, when crawling the Linked Data, we noted that a considerable fraction of the datahub.io resources has some kind of availability problem. In special, two problems are worth mentioning: (1) when the resource does not exist in the specified URL, or (2) when the server that manages the resource is not able to respond in a reasonable time.

Dist-CrawlerLD has a special tripleset cleanup procedure to eliminate bad resources before any crawling task. The procedure works as follows: for each tripleset indexed by the tool, the tool will verify if it has a valid SPARQL Endpoint or RDF Dump file and, if the dataset does not have any of them, the tool

removes the tripleset. By valid, we mean that the resource must respond to a valid HEAD request in a reasonable time (10 seconds). HEAD requests expect that the server returns only the HTTP Header of a resource, and they are commonly used to verify the availability, the file size, or if the resource has changed since the last request.

From almost 550 triplesets available, DIST-CrawlerLD eliminated over 150 of them using this technique, without sacrificing the end result. Section 7.5.2 will show how this test affected the performance of the crawler.

7.3.3. A Brief Description of the Main Actors

Figure 13 summarizes how the actor model was applied to construct DIST-CrawlerLD. In this thesis, we adopted the Akka Framework[19], an Actor-based runtime for managing concurrency, elasticity, and resilience on the Java Virtual Machine.

CrawlerLDMainActor is the actor responsible for receiving the user input and for managing the final result. It will create several **LevelActor**s, one for each level specified by the user, as explained in Section 4.3. For each resource at a predefined level, **CrawlerLDMainActor** will send a *CalculateResource* message to the corresponding **LevelActor**. This actor is just responsible for indicating when a level is finished.

For each *CalculateResource* a **LevelActor** receives, it will create one **ResourceActor** to handle the resource. The **ResourceActor** will identify and create every processor that is eligible to run in the task (each processor instance is also an actor in the DIST-CrawlerLD architecture). The **ResourceActor** will send a *Calculate* message for each Actor Processor (DefererenceProcessor, NumberOfInstancesProcessor or PropertyQueryProcessor). Note that one processor will be represented by an actor for each resource specified.

A processor will execute its task and, once finished, will send a *ResourceProcessed* message back to **ResourceActor,** with the data crawled. The **ResourceActor**, as a state machine, will process the *ResourceProcessed* message and wait for the others processors. As soon as all processors send their *ResourceProcessed* messages, the **ResourceActor** will send a *ResourceProcessed*

[19] http://akka.io/

message to the **LevelActor**, which will simply pass it to **CrawlerLDMainActor**, using the *ResourceProcessedFromLevel* message. The **CrawlerLDMainActor** will merge its current state with the new information available inside the message and will make it available to the user. Once all resources from the level are processed, the **LevelActor** will send the *LevelFinished* message to the **CrawlerLDMainActor** to evaluate which resources will be at the next level, repeating the process until no more resources are available, or the maximum number of resources or the maximum number of levels parameters are reached.

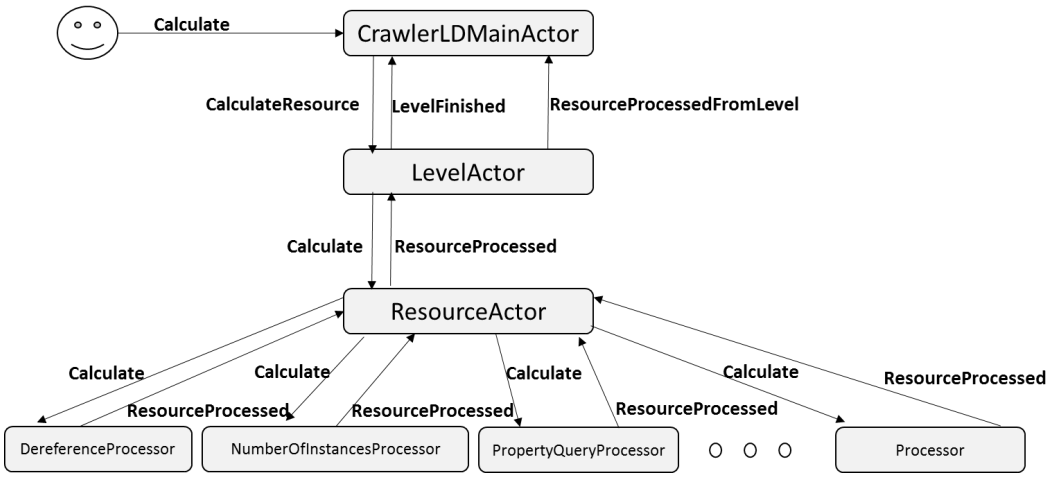

Figure 13. CrawlerLD actors message exchange.

Figure 14 shows how the three processors currently implemented – **DereferenceProcessor**, **NumberOfInstancesProcessor** and **PropertyQueryProcessor** – extract information from the LOD cloud. Note that, in Figure 14, these actors will be collectively referred to as *processor* whereas, in Figure 13, they retain their original name.

The **SparqlQuerierMasterActor** is a special type of actor, since only one instance of this actor exists in the tool. It manages a pool of **SparqlQuerierActor** instances to avoid a large number of locked threads. Since Apache Jena only makes synchronous calls to the endpoints, we had to introduce this actor to handle multiple threads. The **SparqlQuerierMasterActor** also implements a balancing pool (see Section 7.3.4).

Once a processor receives a *Calculate* message, it will formulate a SPARQL query and send it to the **SparqlQuerierMasterActor** with a list of datasets that it wants to crawl. The **SparqlQuerierMasterActor** will start one **SparqlQuerierActor** for each chosen dataset using the *ProcessSparqlOnDataset*

message. Once it receives the resultset from an input dataset, the **SparqlQuerierActor** will send a *SparqlResultset* message to the **SparqlQuerierMasterActor**, which will pass it to the processor and check if any dataset is missing. As soon as the last **SparqlQuerierActor** returns its result, the **SparqlQuerierMasterActor** will send the *QueryFinishedMessage* message to the processor; the message also includes information about any error that happened during the querying process.

We stress that, from the performance perspective, the major difference between DIST-CrawlerLD and CrawlerLD lies in how a processor handles messages. In CrawlerLD, a processor waited for all results before sending data back to the crawler. By contrast, in DIST-CrawlerLD, a processor keeps getting new data, processing it and eliminating what is not useful anymore. The *Jena* resultset (the culprit of the large memory footprint) is handled inside the *SparqlResultset* message and is eliminated by a processor right after it receives the *SparqlResultSset* message.

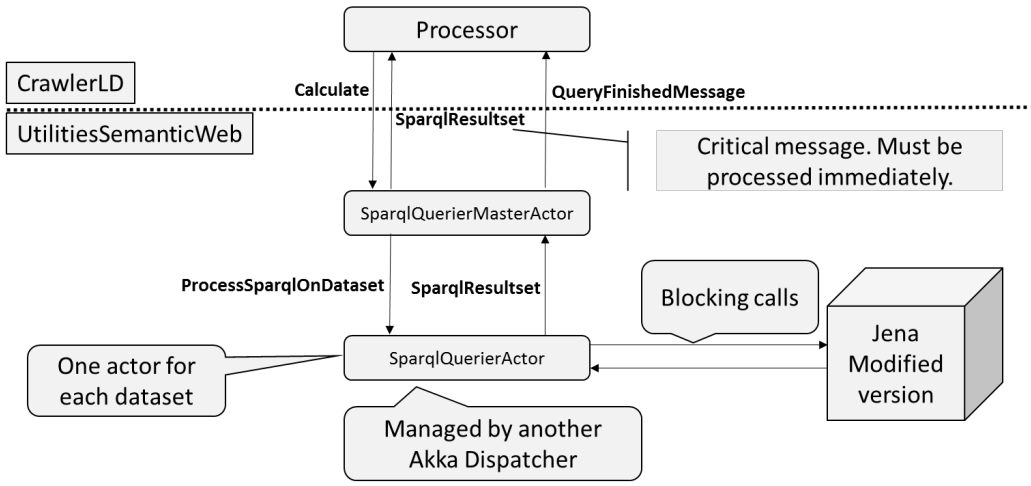

Figure 14. Utilities Semantic Web actors message exchange.

7.3.4. Controlling Distributed Crawling

In this section, we briefly comment on how messages are distribute to remote actors in DIST-CrawlerLD.

The Akka framework provides a simple round-robin algorithm that chooses which actor will receive the next message. This solution is suitable for tasks that spend the same average time to conclude. However, this is not the case for DIST-CrawlerLD: each dataset may have different latencies and resultsets. In the worst

case scenario, the tasks that consume more time may be sent to the same actor so that, while other actors may be idle, a single actor may be assigned to an increasing queue of tasks.

Another approach is to create a customized *balancing pool*. Briefly, using a balancing pool, the mailbox of each remote actor will always be empty and the actor will receive a new request message on-demand: once it finishes a task, it will receive a new one. All messages are stored in an internal queue of the pool, which will be consumed as remote actors select the available tasks.

While the balancing pool has a better performance for the worst-case scenario, the round-robin algorithm wins on the best-case scenario, since each actor will typically have a message to process. Using the balancing pool, an actor may have to wait between the *SparqlResultset* and *ProcessSparqlOnDataset* messages (the latency will increase in a distributed environment, which implies that a SparqlQuerierActor may stay idle for some time before receiving a new calculation message). A possible solution would be to guarantee that at least two messages are available for each actor, a strategy not explored by the current implementation of DIST-CrawlerLD.

Recall that *serialization* is the process of taking objects and converting their state information into a form that can be stored or transported. In the current implementation, while most messages were automatically serialized, the *SparqlResultset* message had to be changed as it contains the resultset. The change focused on creating another *SparqlResultset* message that caches the entire resultset and enables serialization. This change could affect the overall performance and is only enabled if the configuration "*distributionEnabled"* is set to true.

To conclude, Table 10 shows all distribution-aware parameters of DIST-CrawlerLD.

Table 10. **Distribution aware parameters**

Parameter	Description
distributionEnabled	Enables the tool to serialize the messages between remote machines. Is must be enabled to allow distributed computing or some messages will fail to serialize.
useRouter	Indicates if the system should use a built-in router of a custom balancing pool. The balancing pool is recommended in distribution environments.
numberOfActors	Used by the distributed balancing pool, indicates how many **SparqlQuerierActor** should be create.
remoteActors	A list of crawlerLD.distribution modules that will equally receive the actors. Kept empty to run locally.

7.3.5. External Interfaces

CrawlerLD.gui is the module responsible for responding to Web requests sent to DIST-CrawlerLD. It is engineered to be very simple to deploy, as it relies on some **CrawlerLD.core** and **USW classes** and can be used with the **CrawlerLD.distribution** module. Any user should be capable to use DIST-CrawlerLD by issuing only one command-line instruction.

Recall that the *Representational State Transfer* (REST - Fielding et a., 2002) style is an abstraction of the architectural elements within a distributed hypermedia system (Fielding and Taylor 2002). A *REST Service* is a service that responds to a HTTP Request, this request may have a complex structure and may expect another complex structure as response. Table 11 shows the commands available to any system that wants to use DIST-CrawlerLD.

Table 11. **REST Commands available.**

Path	Description
/datasets	List all datasets available to crawl over the Linked Data.
/processors	List all processors available to use in the crawling task.
/tasks	Show all tasks ran or that are being runed by the tool. It gives little information about each task, sufficient to know if it is still running, how much time it spent, and how many URI resources it has already probed.
/taskDetail	Gives all details of a specified task: the crawling, probed, and found resources, and the relationships and provenance of each finding. Shows all parameters specified at the beginning of the task, how much time each resource spent on the crawling task, and many more information.

/newTask	Start a new crawling task. It receives the datasets to be crawled, the processors to be used and the parameters specified in section 4.3.

The current user interface is just an example of what is possible to implement using the REST Service. It was designed to facilitate the performance evaluation tasks. A demo is available at `http://crawlerld-service.cloudapp.net:1002/`. The rest of this section shows examples of the user interface.

Figure 15 shows how a user can create a new task to the crawler. The user have to set the initial resources and parameters (as shown in section 4.3), and they are allowed to define which processor will be used as well as which datasets will be crawled. After clicking "new task button", the tool will show a list of tasks that were processed, or that are being processed (Figure 16). The screen shows only the most relevant information of a task, such as its identifier, current status, current level, number of resources probed until the moment, and its start time and last update time.

Clicking in "more details", the crawler will expose all details available about the selected task (Figure 17 and Figure 18). At this screen, the user is able to evaluate all parameters of the task and how the task performed. The user can select each one of the crawled resources to observe which resources 'found' the selected one and which were found by the resource itself. In addition, it is possible to identify in which property each resource was found.

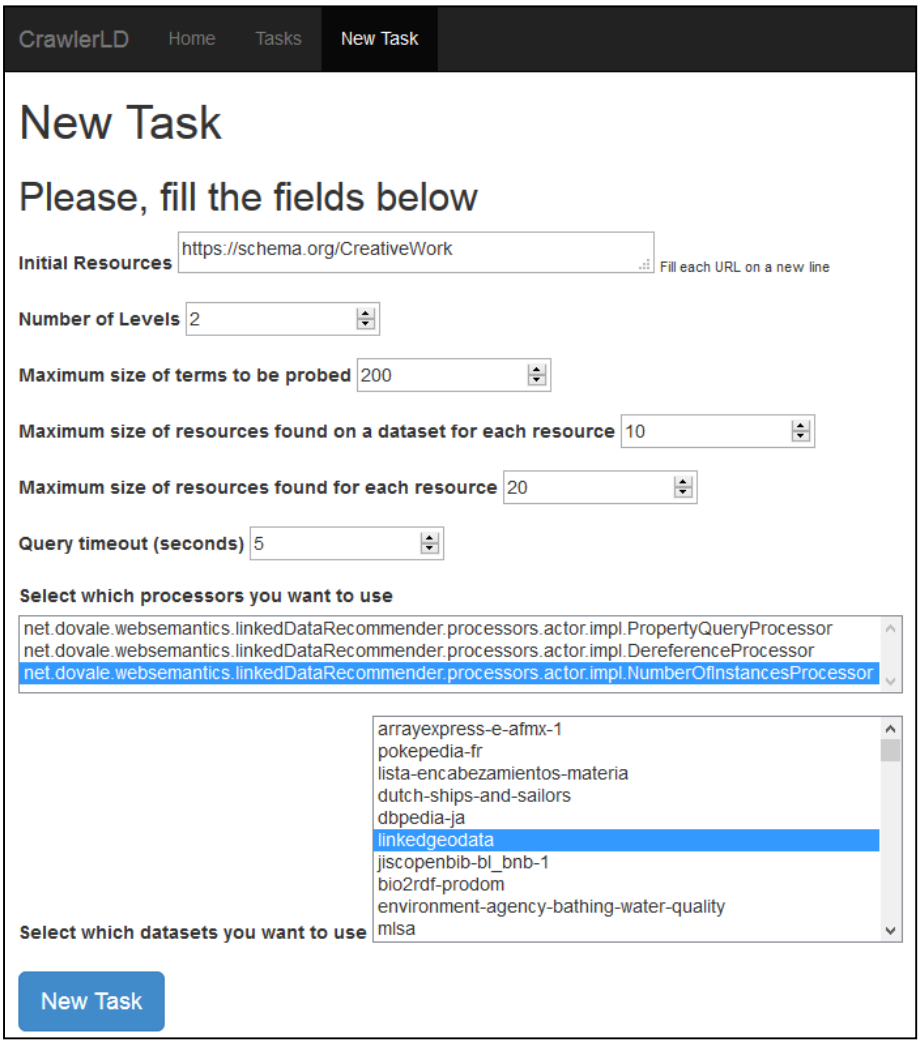

Figure 15. Creating a new crawling task

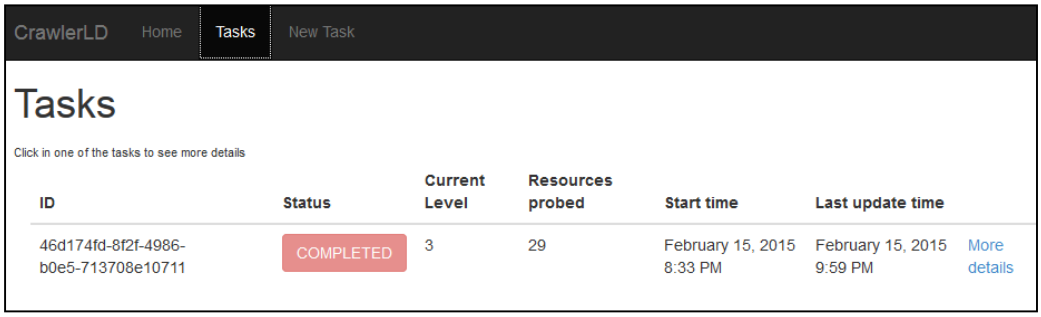

Figure 16. List of tasks

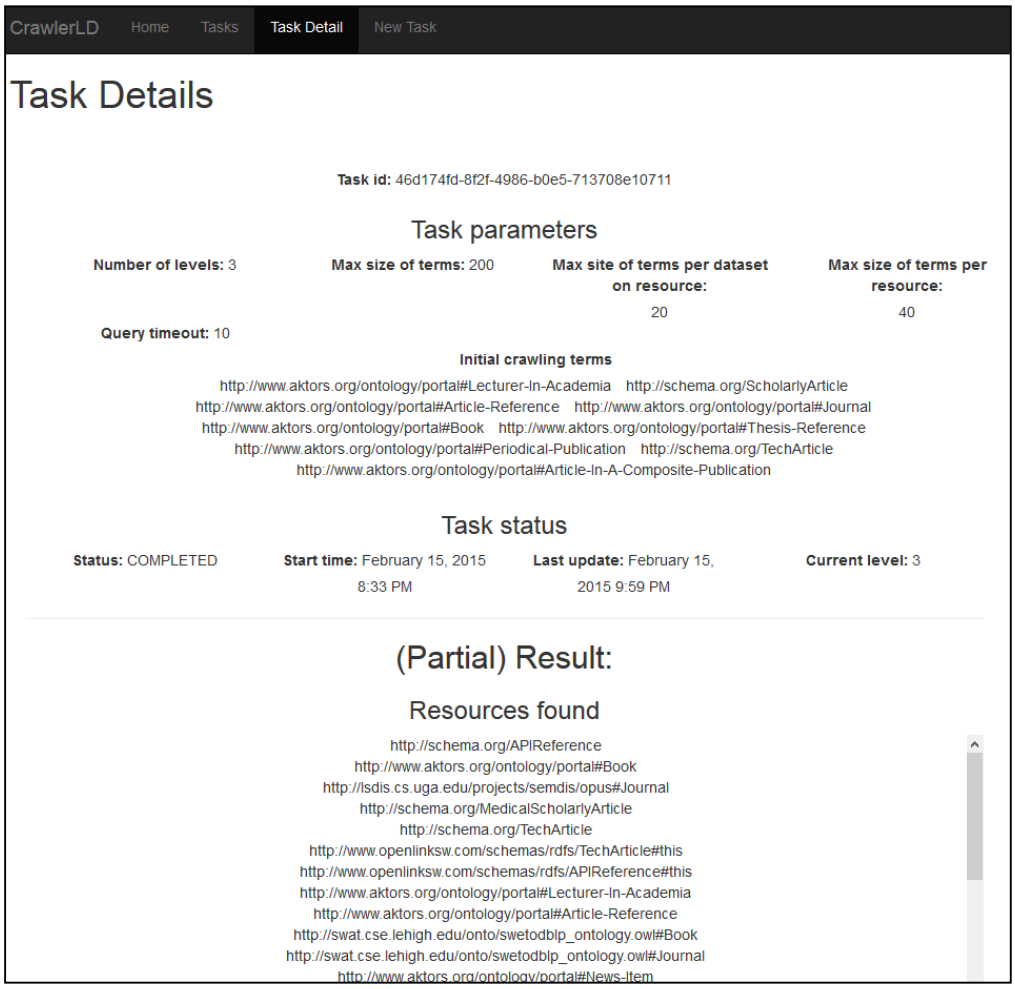

Figure 17. Crawling Task detail (part 1/2)

Figure 18. Crawling Task detail (part 2/2)

7.4. Experiments

The experiments were organized as in Section 6.4.1. However, since DIST-CrawlerLD was able to crawl deeper into the LOD cloud, we repeat the description of the experiments with the new crawling results for the music and publications domains. This became necessary since there was a time interval of over six months between the CrawlerLD and DIST-CrawlerLD evaluations.

In the examples that follow, we used the abbreviations shown in Table 12.

Table 12. **Namespace abbreviation**

Abbreviation	Namespace
akt	http://www.aktors.org/ontology/portal#
dbpediaOntology	http://dbpedia.org/ontology/
dbpediaResource	http://dbpedia.org/resource/
dbpediaYago	http://dbpedia.org/class/yago/
mo	http://purl.org/ontology/mo/
nerdeurocom	http://nerd.eurecom.fr/ontology#
opencyc	http://sw.opencyc.org/concept/
opencycJune2008	http://sw.opencyc.org/2008/06/10/concept/
schema	http://schema.org/
umbel	http://umbel.org/umbel/sc/
openlinksw	http://www.openlinksw.com/schemas/rdfs/
W3	http://www.w3.org/ns/ma-ont#
swcyc	http://sw.cyc.com/concept/
cseLehigh	http://swat.cse.lehigh.edu/onto/swetodblp_ontology.owl#
Lsdis	http://lsdis.cs.uga.edu/projects/semdis/opus#
knowledgeweb	http://knowledgeweb.semanticweb.org/semanticportal/OWL/Documentation_Ontology.owl#
bibTeX	http://purl.org/net/nknouf/ns/bibtex#
Swportal	http://sw-portal.deri.org/ontologies/swportal#
ontoware	http://swrc.ontoware.org/ontology#
marcont	http://www.marcont.org/ontology/marcont.owl#

Results for the Music Domain

We decided to repeat the same evaluation to: (1) validate that DIST-CrawlerLD was capable of recovering similar resources or more resources than CrawlerLD; (2) compare the performance (processing time and resource consumption) of both implementations in the same scenario; (3) evaluate how the LOD Cloud changed

in six months (the time elapsed between the CrawlerLD and DIST-CrawlerLD evaluations).

The initial crawling terms were the same as in Section 5.4.2.

```
mo:MusicArtist
mo:MusicalWork
mo:Composition
dbpedia:Album
dbpedia:MusicalArtist
dbpedia:Single
dbpedia:MusicalWork
dbpedia:Song
wordnet:synset-music-noun-1
```

In what follows, we will first comment on the results obtained in Level 0, for each initial term. Then, we will proceed to discuss how the new terms obtained in Level 0 were processed at Level 1, creating the set of terms in Level 2 (which was not processed by DIST-CrawlerLD due to parameter restriction).

Briefly, DIST-CrawlerLD was able to discover more resources from different ontologies using less resources and time. Table 13 to Table 20 shows the resources found by each crawling resource, the results marked in **bold** were not discovered by CrawlerLD, six months from the time of this evaluation.

Table 13 shows the result of processing the initial crawling term `mo:MusicArtist`. DIST-CrawlerLD found around 2 million instances in the LOD cloud and additional resources to be processed at Level 1. These results clearly are specializations of `mo:MusicArtist` and sum over 1.2 million instances. In addition, a new ontology, `umbel`, was found. Level 2 includes all resources related to what was found in Level 1. Moreover, two new ontologies, `openlinksw` and `opencyc`, were found.

Table 11 shows the result of processing the initial crawling term `mo:MusicalWork`. DIST-CrawlerLD found over 800 thousand instances. At Level 1, DIST-CrawlerLD did not found any new resource, but it discovered a new metadata relationship with other ontologies.

Table 15 shows the result of processing the initial crawling term dbpedia:MusicalWork. DIST-CrawlerLD found seven resources from DBpedia and one from Opencyc. At Level 2, DIST-CrawlerLD was able to find a mixture of new resources and ontologies such as: dbpediaResource, another namespace from DBPedia that represents, not definitions, but pages in Wikipedia, and other 4 ontologies, nerdeurocom, schema, w3 and swcyc.

Table 16 shows the result of processing dbpedia:Song. DIST-CrawlerLD found the most diversified results in terms of query types and query results. It was able to identify resources in different languages (such as Portuguese and Greek), which was only possible because it focused on metadata. Crawlers that use text fields (Nikolov and d'Aquin, 2011) can only retrieve data in the same language as that of initial terms.

Table 17 shows the result of processing dbpedia:Album. DIST-CrawlerLD was able to find again instances in different languages. It also found other ontologies: nerdeurocom, w3, schema, opencyc and swcyc. The resource opencyc:Mx4rwLmi3JwpEbGdrcN5Y29ycA refers to a definition of Album in opencyc ontology and returned 284 instances.

Table 18 shows the results of processing dbpedia:MusicalArtist. The processing of this term exhibited results similar to those obtained by processing dbpedia:Album, in terms of quantity of subclasses. Therefore, it was possible to recover results in multiple languages. It is interesting to observe that some new classes were not available in older experiments. For example, all subclasses of dbpediaOntology:MusicalArtist (such as dbpediaOntology:Instrumentalists) did not previously exist, which indicates that dbpediaOntology is continuously evolving.

Table 19 shows the results of Level 0 for dbpediaOntology:Single. As for other resources from dbpediaOntology, the crawler was able to find a large number of subclasses from opencyc tripleset. In addition, it found more than 160 thousand instances from different triplesets in many languages.

Table 20 shows the results of processing dbpediaOntology:Single. As for other resources from dbpediaOntology, DIST-CrawlerLD was able to find a large number of subclasses from the opencyc dataset. In addition, it found more than 160 thousand instances from different datasets in many languages.

Table 20 shows two resources is which it processing stopped at the first level: `mo:Composition` and `wordnet:synset-music-noun-1`. While previous resources from `Music Ontology` (`mo` namespace) showed us that the ontology is indeed used by other datasets, `Wordnet` seems not to be used for the music domain.

Table 13. **mo:MusicArtist result**

Level	Resource	Instances
1	`mo:MusicArtist`	**1.940.977**
From *uriburner, openlink-lod-cache, musicbrainz, data-open-ac-uk, dbtune-musicbrainz*		
2	`mo:MusicGroup`	449.962
2	`mo:SoloMusicArtist`	835.219
2	**`umbel:MusicPerformanceAgent`**	0
3	**`openlinksw:MusicGroup#this;`** **`umbel:Band_MusicGroup;`** **`umbel:MusicalPerformer;`** **`opencycJune2008:en/MusicPerformanceAgent;`** **`opencycJune2008:Mx4rwDSivJwpEbGdrcN5Y29ycA;`** **`opencycJune2008:Mx4rwDSivJwpEbGdrcN5Y29ycA;`** **`opencyc:Mx4rwDSivJwpEbGdrcN5Y29ycA`**	0

Table 14. **mo:MusicalWork result**

Level	Resource	Instances
1	`mo:MusicalWork`	**797.921**
From *rkb-explorer-foreign, openlink-lod-cache*		
2	`mo:Movement;` **`umbel:AudioConceptualWork`**	0
3	**`umbel:Multi_MovementComposition;`** **`opencyc:Mx4rwAXXLZwpEbGdrcN5Y29ycA;`** **`opencycJune2008:Mx4rwAXXLZwpEbGdrcN5Y29ycA`**	0

Table 15. dbpediaOntology:MusicalWork result

Level	Resource	Instances
1	dbpediaOntology:MusicalWork	**794.498**
From *dbpedia-eu, dbpedia-de, dbpedia-fr, uriburner, sztaki-lod, dbpedia-nl, dbpedia-live, openlink-lod-cache, dbpedia, dbpedia-pt, dbpedia-el, dbpedia-ja*		
2	dbpediaOntology:ArtistDiscography	9.716
2	dbpediaOntology:Song	56.915
2	**dbpediaOntology:NationalAnthem**	0
2	dbpediaOntology:Opera	4.409
2	**dbpediaOntology:ClassicalMusicComposition**	1.184
2	dbpediaOntology:Single	212.183
2	dbpediaOntology:Album	656.198
2	**opencyc:Mx4rwAXXLZwpEbGdrcN5Y29ycA**	0
3	**dbpediaResource:Rota; dbpediaResource:Chant; dbpediaResource:Balisong; dbpediaResource:SMP; dbpediaResource:Mater; dbpediaResource:Folksong; dbpediaOntology:EurovisionSongContestEntry; dbpediaResource:KALI; dbpediaResource:Songs; dbpediaResource:Een; dbpediaResource:CRY; nerdeurocom:Song; dbpediaResource:Song; opencyc:Mx4rwP3teJwpEbGdrcN5Y29ycA; schema:MusicRecording#this; dbpediaResource:Popera; dbpediaResource:Operas; dbpediaResource:Opera; dbpediaResource:Cd-single; dbpediaResource:CD-single; dbpediaResource:Singles; dbpediaResource:Single; opencyc:Mx4rv6i4pJwpEbGdrcN5Y29ycA; nerdeurocom:Album; dbpediaResource:Albums; w3:Collection; dbpediaResource:Studioalbum; dbpediaResource:Album; opencyc:Mx4rwLmi3JwpEbGdrcN5Y29ycA; schema:MusicAlbum#this; umbel:AudioConceptualWork; swcyc:Mx4rwAXXLZwpEbGdrcN5Y29ycA; opencyc:Mx4rwL5Y-5wpEbGdrcN5Y29ycA; opencyc:Mx4rvrPdMZwpEbGdrcN5Y29ycA; opencyc:Mx4rPzqQQitqEdiaugAH6RYvVQ; opencyc:Mx4rwUwN3ZwpEbGdrcN5Y29ycA; opencyc:Mx4rwVOgtJwpEbGdrcN5Y29ycA; opencycJune2008:Mx4rwAXXLZwpEbGdrcN5Y29ycA**	0

Table 16. dbpediaOntology:Song result

Level	Resource	Instances
1	dbpediaOntology:Song	**28.698**
From *dbpedia-de, uriburner, sztaki-lod, dbpedia-nl, yovisto, dbpedia-live, dbpedia, dbpedia-pt, dbpedia-el*		
2	dbpediaOntology:EurovisionSongContestEntry	5.288
2	**nerdeurocom:Song**	0

2	**opencyc:Mx4rwP3teJwpEbGdrcN5Y29ycA**	17
2	schema:MusicRecording#this	0
3	**umbel:Song_CW;** **swcyc:Mx4rwP3teJwpEbGdrcN5Y29ycA;** **opencyc:Mx8Ngx4rwEcGC5wpEbGdrcN5Y29ycB4rwKrQ** **NpwpEbGdrcN5Y29ycB4rwP3teJwpEbGdrcN5Y29ycA;** **opencyc:Mx4rvVjPBZwpEbGdrcN5Y29ycA;** **opencyc:Mx4r_3NStEeEEdaAAABQ2sS97g;** **opencyc:Mx4rwAzWmpwpEbGdrcN5Y29ycA;** **opencyc:Mx4rv49v0pwpEbGdrcN5Y29ycA;** **w3:Track**	0

Table 17. **dbpediaOntology:Album result**

Level	Resource	Instances
1	dbpediaOntology:Album	**400.473**
From *dbpedia-eu, dbpedia-de, dbpedia-fr, uriburner, sztaki-lod, dbpedia-nl, dbpedia-live, dbpedia, dbpedia-pt, dbpedia-el, dbpedia-ja*		
2	**nerdeurocom:Album; w3:Collection;** schema:MusicAlbum#this	0
2	**opencyc:Mx4rwLmi3JwpEbGdrcN5Y29ycA**	284
3	**swcyc:Mx4rwLmi3JwpEbGdrcN5Y29ycA;** **umbel:Album_CW;** **opencyc:Mx4rsuAeiOYRQdaV9rYefTiDdQ;** **opencyc:Mx4rrCLPHuYRQdaYGdGb966k4g;** **opencyc:Mx4rpC0M3uYRQdaMgfF3U2mGqQ**	0

Table 18. dbpediaOntology:MusicalArtist result

Level	Resource	Instances
1	dbpediaOntology:MusicalArtist	**176.265**
From *dbpedia-eu, dbpedia-de, dbpedia-fr, uriburner, sztaki-lod, dbpedia-nl, dbpedia, dbpedia-pt, dbpedia-el, dbpedia-ja*		
2	**dbpediaOntology:Instrumentalist**	2.718
2	**dbpediaOntology:BackScene**	225
2	**dbpediaOntology:MusicDirector;** **dbpediaOntology:Singer**	0
2	**dbpdiaOntology:ClassicalMusicArtist**	671
2	**opencyc:Mx4rvVisB5wpEbGdrcN5Y29ycA**	482
3	**dbpediaOntology:Guitarist;** **dbpediaYago:Musician;** **umbel:Musician;** **swcyc:Mx4rvVisB5wpEbGdrcN5Y29ycA;** **opencyc:Mx4rvVjp3ZwpEbGdrcN5Y29ycA;** **opencyc:Mx4rvVjqXpwpEbGdrcN5Y29ycA;** **opencyc:Mx4r7wQOfEeAEdaAAABQ2sS97g;** **opencyc:Mx4rHV7wICwxQdiRQdQSzSL6Dw;** **opencyc:Mx4rzQXDcip_QdiIBoeuUEmDxA;** **opencyc:Mx4rPzreYitqEdiaugAH6RYvVQ;** **opencyc:Mx4rCThRjlILEdqAAAACs71DGQ**	0

Table 19. dbpedia:Single result

Level	Resource	Instances
1	dbpediaOntology:Single	**161.047**
From *dbpedia-de, uriburner, sztaki-lod, dbpedia-nl, dbpedia-live, dbpedia, dbpedia-pt, dbpedia-el, dbpedia-ja*		
2	**opencyc:Mx4rv6i4pJwpEbGdrcN5Y29ycA**	45
3	**swcyc:Mx4rv6i4pJwpEbGdrcN5Y29ycA**	0

Table 20. mo:Composition and wordnet:synset-music-nount-1 results

Level	Resource	Instances
1	mo:Composition	**796.954**
From *openlink-lod-cache*		
1	wordnet:synset-music-noun-1	**0**

Results for the Publications Domain

As already pointed out, we decided to repeat the same evaluation to: (1) validate that DIST-CrawlerLD was capable of recovering similar resources or more resources than CrawlerLD; (2) compare the performance (processing time and resource consumption) of both implementations in the same scenario; (3) evaluate how the LOD Cloud changed in six months (the time elapsed between the CrawlerLD and DIST-CrawlerLD evaluations).

The initial crawling terms were the same as in Section 5.4.2:

```
schema:TechArticle
schema:ScholarlyArticle
akt:Article-Reference
akt:Article-In-A-Composite-Publication
akt:Book, akt:Thesis-Reference akt:Periodical-Publication
akt:Lecturer-In-Academia
akt:Journal
```

Table 21 summarizes the results. CrawlerLD and the basic version were not able to find complex results for these initial crawling terms. However, six months after the last evaluation, DIST-CrawlerLD found some new resources from other ontologies (marked in **bold**). These new discovered terms indicate that the ontologies used in this evaluation are becoming popular. New evaluations in the future might confirm this trend.

Table 21. Publication domain results

Level	Resource	Instances
1	schema:TechArticle	**1**
From *uriburner*		
2	schema:APIReference; openlinksw:TechArticle#this	0
3	**openlinksw:APIReference#this**	0
1	schema:ScholarlyArticle	**103**
From *uriburner*		
2	schema:MedicalScholarlyArticle; **openlinksw:ScholarlyArticle**	0
3	**openlinksw:MedicalScholarlyArticle#this**	0
1	akt:Article-Reference	**1.218.625**
From *rkb-explorer-deepblue, rkb-explorer-budapest, rkb-explorer-roma, rkb-explorer-newcastle, rkb-explorer-laas, rkb-explorer-deploy, rkb-explorer-pisa, rkb-explorer-ibm, rkb-explorer-irit, rkb-explorer-curriculum, rkb-explorer-ulm, rkb-explorer-risks, rkb-explorer-ft, rkb-explorer-eurocom, dbpedia-pt, openlink-lod-cache*		
1	akt:Article-In-A-Composite-Publication	**0**
2	akt:News-Item	0
1	akt:Thesis-Reference	**603**
From *rkb-explorer-newcastle, rkb-explorer-laas, rkb-explorer-ibm, rkb-explorer-irit, rkb-explorer-eurocom, rkb-explorer-ulm, openlink-lod-cache*		
1	akt:Book	**0**
2	**cseLehigh:Book; lsdis:Book**	0
3	**knowledgewe:Book; bibTeX:Book; swportal:Book; ontoware:Book; marcont:Book; cseLehigh:Edited_Book; lsdis:Edited_Book**	0
1	akt:Periodical-Publication	**0**
2	akt:Newspaper	0
3	akt:Daily-Newspaper	0
1	akt:Lecturer-In-Academia	**37**
From *rkb-explorer-newcastle*		
1	akt:Journal	**34.212**
From *rkb-explorer-deepblue, rkb-explorer-budapest, rkb-explorer-roma, rkb-explorer-newcastle, rkb-explorer-lass, rkb-explorer-deploy, rkb-explorer-pisa, rkb-explorer-ibm, rkb-explorer-irit, rkb-explorer-ulm, rkb-explorer-kaunas, rkb-explorer-eurocom, dbpedia-pt, openlink-lod-cache*		
2	**cseLehigh:Journal; lsdis:Journal**	0
3	**swportal:Journal**	0

7.5. A Performance Comparison with Previous Implementations

In this section, we compared DIST-CrawlerLD with CrawlerLD (of Chapter 6) and the *basic implementation* (of Chapter 5).

7.5.1. Resources Consumption Analysis

Figure 19 and Figure 20 show the plots of the percentage of CPU time and the amount of memory used when processing a single task, respectively, for CrawlerLD and DIST-CrawlerLD. In both cases, the same initial resource (`dbpedia:Song`) and the same parameters were used (2 levels and approximately 500 datasets). Both tasks ran in an Azure[20] virtual machine with 8 cores and 56GB of memory (STANDARD_A7 azure instance), which was provided by Azure for Research Program[21].

We note that the X axes of the plots for CrawlerLD show only in the first 10 minutes, while those for DIST-CrawlerLD show in the first hour. DIST-CrawlerLD took longer to finish as it found more resources and crawled Level 2 (it found 227 resources, 16 were processed on both first levels – 0 and 1 - and the other 211 resources, although eligible to be probed at next level, were not); by contrast, the CrawlerLD did not find any new resource and stopped at Level 0, crawling just one resource.

Also, the Y axes of the memory utilization plots are in different scales. Both versions were configured to use at most 11 GB. However, while CrawlerLD reached a maximum of 8 GB, DIST-CrawlerLD used only 3 GB. In fact, CrawlerLD used much more memory than DIST-CrawlerLD, as expected. While DIST-CrawlerLD hit 2.5 GB of memory use only at the end of Level 1 (when processing over 15 resources), CrawlerLD hit 7 GB right at Level 0 (just 1 resource).

Another indicator of how DIST-CrawlerLD uses machine resources in a healthier way is how the memory utilization increases. The Java Virtual Machine (JVM) uses garbage collection (GC) to identify and free unused memory resources. From time to time, and when JVM runs out of resources, JVM initiates

[20] http://azure.microsoft.com/

[21] http://research.microsoft.com/en-us/projects/azure/

GC to try to free up memory. Since the execution of GC is expensive, JVM uses a greedy algorithm to decide when to start GC: if there is memory available, avoid its execution, if the current maximum available is reached, execute GC and try to increase the maximum memory available by asking for more memory from the operating system.

The memory utilization plot shows the behavior of both implementations and how the GC behaved in each case. For CrawlerLD, memory is always increasing, even when GC is executed (when the memory use is reduced). After each GC execution, memory allocation always increases. For DIST-CrawlerLD, memory allocation is also always increasing but, after GC execution, the system returns to the same memory utilization level as before. The maximum memory increases since JVM finds that it can increase memory to reduce the frequency with which GC is called. In previous experiments, we found that 2 GB of maximum memory allocation was sufficient to an 8 CPU core machine.

The percentage of CPU utilization is another health indicator. While CrawlerLD used 50% of CPU time, on the average, DIST-CrawlerLD stayed at 10%, on the average. The high percentage of CPU use observed for CrawlerLD could be related to how many times GC was called. On DIST-CrawlerLD, the CPU idle behavior is expected as the system spent a considerable amount of time waiting for triplesets responses.

Another important observation is how many queries each version ran. Recall from Section 6.3 that DIST-CrawlerLD executes:

- Dereference processor: 1 query for each resource.
- Instance Counter processor: 1.5 query for each resource and dataset (half of the datasets returned error when trying to group the number of instances, which forced our processor to run a simpler query).
- Property processor: 1 query for each resource and dataset.
- The same applies to CrawlerLD. Consequently, for each crawling resource, both tools executed:

Number of queries per resource crawled = 1 + (1.5 * D) + D = 1 + 2.5 * D

where *D* is the number of datasets. Since *D* is approximately 500 in the experiments, the tools made approximately 1,251 queries per resource crawled.

Furthermore, in the experiments, CrawlerLD crawled just one term, whereas DIST-CrawlerLD crawled a total of 16 terms. Hence, we have:

- Number of queries executed (CrawlerLD) = 1,251
- Number of queries executed (DIST-CrawlerLD) = 20,016

Even so, comparing the maximum memory utilization, DIST-CrawlerLD spent 2.8 times less resource than the older version.

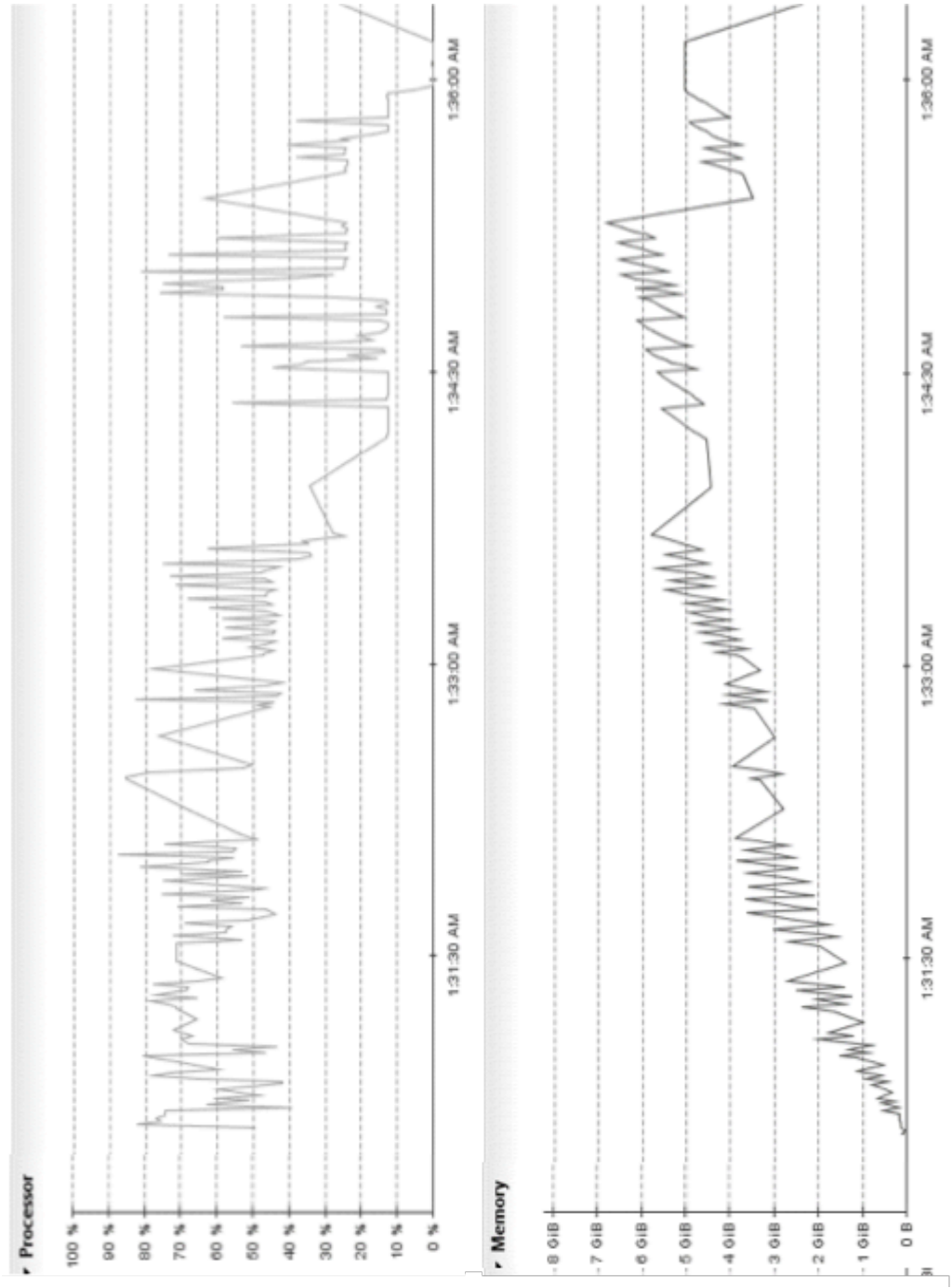

Figure 19. CrawlerLD execution pattern.

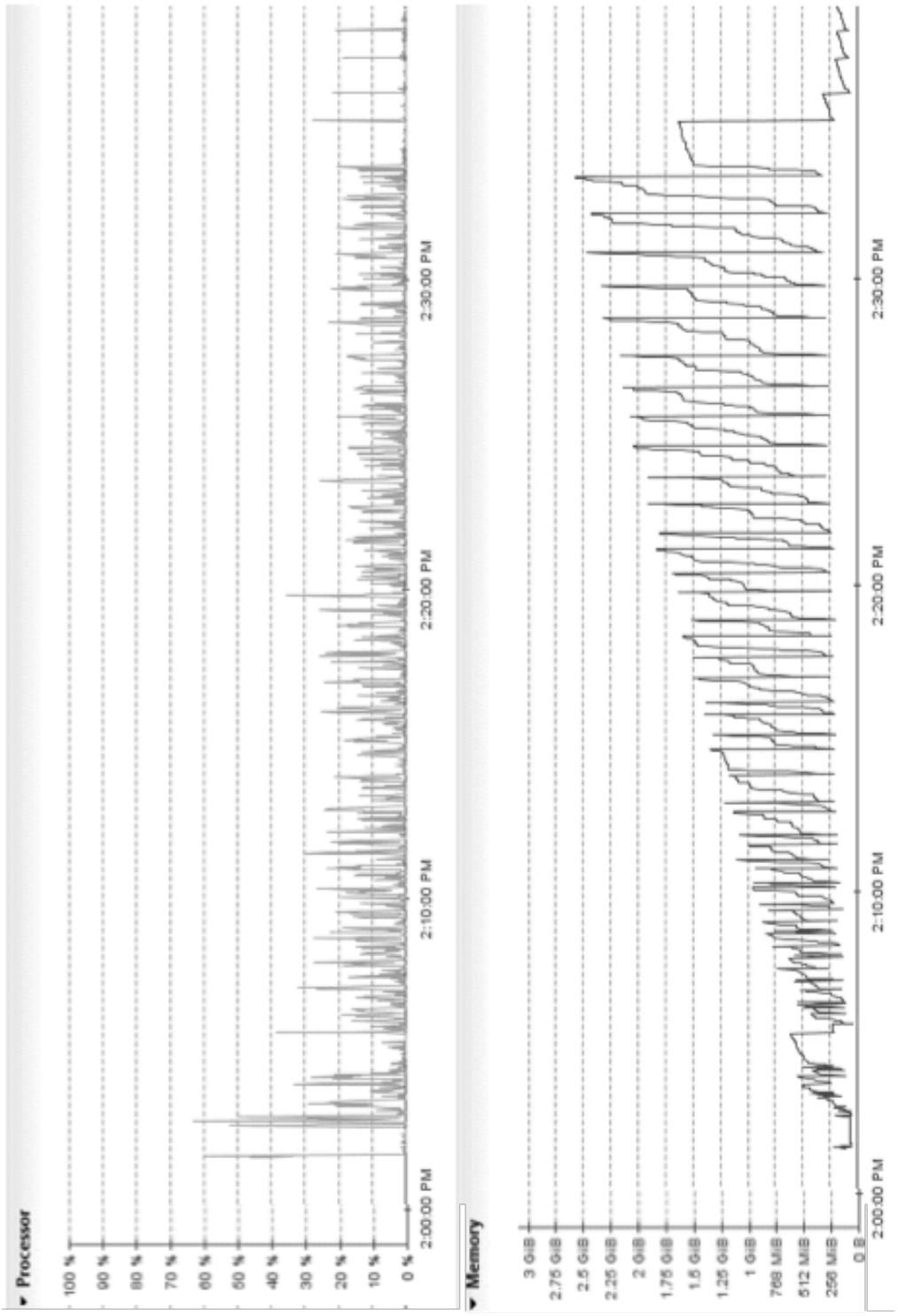

Figure 20. DIST-CrawlerLD execution pattern.

7.5.2. Processing Time Analysis

In this section, we evaluate the time consumed by each one of the three crawlers.

Table 22 and Table 23 show the time needed to process each resource in each implementation and how many resources were probed in each. In addition, we added a comparison of the tool with and without the tripleset availability test presented in section 7.3.2. While the basic implementation spent one order of magnitude more time than the subsequent implementation, CrawlerLD and DIST-CrawlerLD took approximately the same time to process each term in the initial set of terms. However, DIST-CrawlerLD retrieved more resources than CrawlerLD. On the other hand, when using the availability test, which reduced the number of triplesets from almost 550 to 400 (27% less triplesets), the results were retrieved 70% faster on the average. Even with the reduced number of triplesets, this version was able to recover all data retrieved before and, in some cases, recover even more information. Indeed, the *tripleset availability test* is very conservative and removes only the triplesets that will not return results in any case.

Another important fact is that, due to the higher memory consumption of the two older implementations, it was not possible to create an initial crawling term set with more than one element. In our previous experiments, this input created such a high memory footprint that even a 64GB memory machine was not able to handle. DIST-CrawlerLD, on the other hand, was able to process all 9 initial crawling terms of the Music Domain experiment in only one execution, which consumed 4 GB of memory at most.

Table 22. Time consumed (in minutes) for the Music domain

Term	Basic[22]	CrawlerLD	DIST-CrawlerLD	DIST-CrawlerLD (With availability test)[23]
	Time	**Time**	**Time**	**Time**
	Terms crawled / found	*Terms crawled / found*	*Terms crawled / found*	*Terms crawled / found*
mo:MusicArtist	70	11	10	2
	3 / 2	3 / 2	4 / 11	3 / 6
mo:MusicalWork	28	8	8	1
	2 / 1	2 / 1	3 / 5	3 / 5
mo:Composition	14	4	3	< 1
	1 / 0	1 / 0	1 / 0	1 / 0
dbpedia:MusicalWork	183	22	24	4
	30 / +20k[24]	6 / 5	9 / 46	9 / 41
dbpedia:Song	163	11	12	4
	3 / 2	3 / 2	5 / 12	15 / 226
dbpedia:Album	173	8	13	3
	17 / 16	2 / 1	5 / 9	8 / 74
dbpedia:MusicalArtist	167	4	18	7
	15 / +2k	1 / 0	7 / 17	19 / 383
dbpedia:Single	186	4	5	3
	19 / +3k	1 / 0	2 / 2	6 / 36
wordnet:synset-music-noun-1	24	11	3	< 1
	1 / 0	1 / 0	1 / 0	1 / 0
Average per terms crawled (music domain)	9.98 min	4.15 min	2.60 min	0.40 min

[22] The first version used the property rdfs:seeAlso instead of owl:equivalentClass, which returned more results but with less precision.

[23] The evaluation with availability test (see section 7.3.2) occurred in a different timeframe, which justifies the result difference between both versions of DIST-CrawlerLD.

[24] Some crawling tasks returned a larger number of terms, primarily from the yago tripleset. These resources were not discoverable after the execution of the first experiment, on January 2014.

Table 23. **Time consumed (in minutes) for the Publications domain.**

Term	Basic[25]	CrawlerLD	DIST-CrawlerLD	DIST-CrawlerLD (With availability test)
	Time	Time	Time	Time
	Terms crawled / found	*Terms crawled / found*	*Terms crawled / found*	*Terms crawled / found*
schema:TechArticle	29	4	9	1
	N/A	N/A	3 / 3	3 / 3
schema:ScholarlyArticle	47	4	9	1
	N/A	N/A	3 / 3	2 / 2
akt:Article-Reference	14	4	3	1
	N/A	N/A	1 / 0	1 / 0
akt:Article-In-A-Composite-Publication	28	8	5	2
	N/A	N/A	1 / 0	2 / 1
akt:Book	14	5	7	2
	N/A	N/A	3 / 9	2 / 6
akt:Thesis-Reference	14	5	2	< 1
	N/A	N/A	1 / 0	1 / 0
akt:Periodical-Publication	28	4	5	2
	N/A	N/A	2 / 2	2 / 2
akt:Lecturer-In-Academia	14	5	3	< 1
	N/A	N/A	1 / 0	1 / 0
akt:Journal	14	4	7	1
	N/A	N/A	3 / 3	2 / 2
Average per terms crawled (publications domain)	N/A	N/A	2.78 min	0.75 min

[25] The first version used the property rdfs:seeAlso instead of owl:equivalentClass, which returned more results, but with less precision.

To conclude, Table 24 and Table 25 present additional statistics for DIST-CrawlerLD using and not using the tripleset availability test and varying the number of levels.

Table 24. **Results for DIST-CrawlerLD in a complex scenario.**

Initial resources	Number of levels	Resources crawled	Resources found	Time spent (minutes)
9	2	63	756	141
9	3	201	1.083	450
			Average time to process one resource	2,3

Table 25. Results applying tripleset availability test

Initial resources	Number of levels	Resources crawled	Resources found	Time spent (minutes)
9	2	64	771	81
9	3	201	1.069	257
			Average time to process one resource	1,27

7.5.3.Distributed Computing Performance

All experiments above ran using a single 8 core machine. To evaluate DIST-CrawlerLD in a distributed environment, we created 11 machines with 2 cores and 3.5 GB of memory (BASIC_A2 Azure instance) each. In addition, the frontend ran into the previous 8 core machine, but it was configured not to not crawl any tripleset.

Table 26 presents the different processing times for each resource when running DIST-CrawlerLD in a single machine and in the distributed environment. Each evaluation was executed five times so that time shown in Table 26 are averages over all executions. Each version returned a similar number of terms to process, but the distributed version spent half of the time on the average. In fact, we observed that the processing time is better when the tool has many resources to process. To prove that, we simulated the same experiment that resulted in Table 26 using all eleven machines. Table 27 and Table 28 present the data we obtained in order to compare with Table 24 and Table 25. While the single machine experiment reached an average of one resource crawled per 2.3 minutes, the distributed experiment achieved an average of one resource crawled per 0.47 minutes. To process 201 resources, the first experiment took more than 7 hours, while the distributed experiment took nearly 1.5 hours. Using the tripleset

availability test, the numbers are even lower: the average time for each crawling resource reduced from 1,27 minutes to 0.14. The total processing reduced from 81 to 7 minutes on two levels, and from 257 to 29 minutes on three levels.

Table 26. Time consumed by actor model single machine and distributed

Term	Single (1) (seconds)	Distributed (1) (seconds)
mo:MusicArtist	118	41
mo:MusicalWork	79	35
mo:Composition	29	16
dbpedia:MusicalWork	214	76
dbpedia:Song	350	118
dbpedia:Album	179	60
dbpedia:MusicalArtist	426	139
dbpedia:Single	145	54
wordnet:synset-music-noun-1	34	18
schema:TechArticle	78	34
schema:ScholarlyArticle	72	35
akt:Article-Reference	33	16
akt:Article-In-A-Composite-Publication	64	30
akt:Book	77	34
akt:Thesis-Reference	34	14
akt:Periodical-Publication	61	31
akt:Lecturer-In-Academia	29	14
akt:Journal	82	42

(1) Average over 5 executions.

Table 27. Additional statistics for DIST-CrawlerLD in a distributed mode

Initial Resources	Number of levels	Resources crawled	Resources found	Time spent (minutes)
9	2	60	611	26
9	3	201	862	96
			Average time to process one resource	0.47

Table 28. Additional statistics applying tripleset availability test

Initial resources	Number of levels	Resources crawled	Resources found	Time spent (minutes)
9	2	61	612	7
9	3	201	862	29
			Average time to process one resource	0.14

7.6.
An Evaluation of the Linked Data Cloud and the Used Ontologies

In this section, the Linked Open Data Cloud will be evaluated in order to demonstrate that datasets classified in the same domain may use several, heterogeneous ontologies, which create a difficult environment to anyone who searches resources of a given domain.

Bizer et al. (Bizer et al., 2014) describe the state of the Linked Open Cloud by August 2014, illustrated in Figure 21. Circles represent triplesets, and arrows indicate relationships between two triplesets. Circles of the same color represent triplesets classified in the same domains (the original diagram unfortunately uses a color code to indicate the classification of triplesets). For example, the green circles on the right of the image are triplesets of the *publications* domain and the purple circles on the top-left part of the image represent triplesets classified in the media triplesets, in which the *music domain* is included.

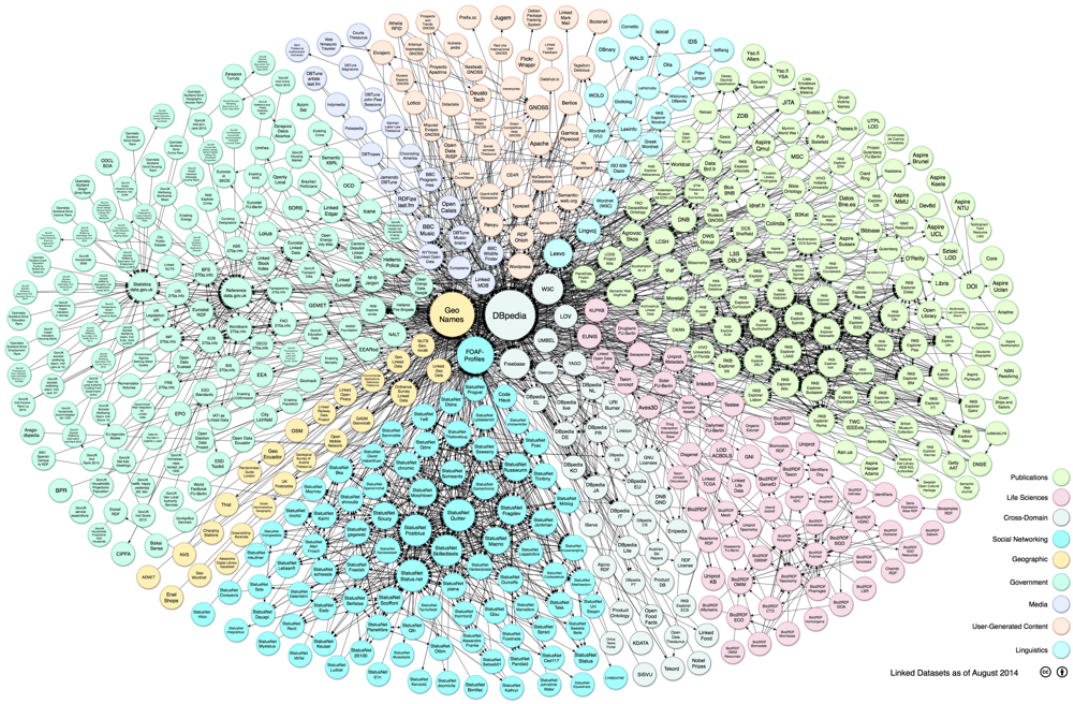

Figure 21. Linked Open Data cloud 2014 state

In this section, our goal is to verify if triplesets classified in the same domain use different ontologies or not. We take the results reported in (Bizer et al., 2014) as the *gold standard* for the purposes of classifying triplesets. We elected the publications domain and retrieved 136 triplesets, out of which 83 triplesets had SPARQL endpoints, while 74 had RDF dumps. Over 103 triplesets had some type

of endpoint (SPARQL or dump files) that DIST-CrawlerLD could process. But, in fact, only 83 were actually available for the current experiment.

The evaluation started by selecting two popular ontologies for the publication domain – Aktors, Dublin Core[26] – and two generic ontologies – DBPedia and Schema.org. In addition, we noted that some triplesets were unable to return a valid result set when the *Property Processor* was executed (see section 6.3) due to the complexity of the *union* query. In particular, all triplesets from *rkb-explorer* did not accept the *union* statement. For this specific evaluation, since we were not interested in time performance, we created a new property processor that works similarly to the *Instance Counter Processor*. First, it applies the complex query (see Figure 8) of over all triplesets available and saves the datasets that return some kind of error. Then it applies the queries introduced in section 4.5 to the triplesets that were unable to return a valid result set for the complex query. This approach is an effort to reach more triplesets without executing too many queries over the endpoints.

The experimental setup was:

- *Universe*: the set of datasets available through the DataHub catalog plus some ontologies (in special, schema.org and its mappings[27]).
- *Gold standard*: the classification reported in (Bizer et al., 2014).
- *Selected domain*: the *publications* domain, with 136 datasets listed in (Bizer et al., 2014); 83 of these datasets actually available for the experiment.
- *Set of initial crawling terms*: the terms listed in the first column of Table 29, selected from Aktors and Dublin Core, two popular ontologies for the *publications* domain, and DBPedia and Schema.org, two generic domain ontologies.
- *Number of levels*: 5.
- *Maximum number of terms probed*: 2000.
- *Maximum number of terms probed for each term in the crawling frontier*: 50.

[26] http://dublincore.org/

[27] http://schema.rdfs.org/mappings.html

- *Maximum number of terms probed in a dataset, for each term in the frontier*: 30.

Table 29 shows the terms used in the evaluation, how many triplesets were found, and the recall based on tripleset availability and precision based on the total amount of triplesets found. The numbers show that the *Aktors* ontology is the most popular ontology for the publications domain: with all resources combined, we were able to find 39% of the triplesets in the domain, with precision of 82%. Dublin Core had worse numbers: 18% of recall and 17% of precision. After grouping both results, a recall of 55% and a precision of 37% was achieved. Furthermore, only one tripleset (msc[28]) uses both ontologies. The generic ontologies (DBPedia and Schema.org), on the other hand, had an insignificant result. From six crawling terms, only one (*Dbpedia:AcademicJournal*) was used by triplesets in the publications domain (in fact, only one tripleset, sztaki-lod[29]).

One should proceed with care to draw conclusions based on the results in Table 29. Indeed, the recall and precision shown in Table 29 reflect several factors, which we highlight, among others:

1 - The number of datasets actually available for the experiment.

2 - The initial set of terms selected.

3 - The number of datasets classified in *publications* domain that indeed use well-known ontologies for the publications domain.

4 - The number of datasets not classified in *publications* domain that use well-known ontologies for the publications domain.

The first point is a limitation of directly crawling the LOD cloud. It could be circumvented by crawling a dump of the LOD cloud, such as the one available from the LOD Laudromatic (Beek et al., 2014).

The second and the third points are interrelated and affect *RLT*, the number of relevant datasets retrieved. Indeed, *RLT* is necessarily sensitive to the set of initial terms and other experiments could be run to further assess this point. But – and this is more important – *RLT* is directly affected by how many datasets classified in the *publications* domain actually use well-known ontologies for the domain. Indeed, we found that only 56% of the datasets classified in the

[28] http://datahub.io/dataset/msc

[29] http://datahub.io/dataset/sztaki-lod

publications domain (and available at the time of the experiment) use well-known ontologies for the domain (i.e., Aktors and Dublin Core). The other 44% of the available datasets use a variety of ontologies: from self-made to less popular.

The fourth point affects *RT*, the number of datasets retrieved. The argument here is symmetric: a dataset *d* may use ontologies that pertain to the *publications* domain (and hence *d* is retrieved), but *d* may not be classified in the *publications* domain. Indeed, the dataset *d* may contain some triples that refer to publications (and which correctly uses ontologies from the *publications* domain), but the main purpose of *d* may not be to store publications and, hence, *d* is not classified in *publications* domain. Based on this argument, the precision of the crawler could be improved by rejecting datasets in which is majority of triples does not belong to the domain in question (an expensive test, unless the catalog contains enough information about the datasets to implement the test).

To conclude, the use of a crawler such as DIST-CrawlerLD to locate datasets that pertain to a given domain is necessarily limited by the adherence of dataset publishers to the Linked Data best practices (Bizer et al., 2009), as expected. Experiments with other domains should be conducted to further assess this conclusion.

Table 29. Publication domain resources and recall

Resource	Triplesets found	Domain triplesets found (1)	Non-Domain triplesets found	Recall (2)	Precision (3)
Domain ontologies					
Aktors:Article-in-A-Composite-Publication	29	27	2	33%	93%
Aktors:Article-Reference	29	26	3	31%	90%
Aktors:Book	22	17	5	20%	77%
Aktors:Journal	29	25	4	30%	86%
Aktors:Lecture-In-Academia	18	17	1	20%	94%
Aktors:Periodical-Publication	29	27	2	33%	93%
Aktors:Research-Interest	33	30	3	36%	90%
Aktors:Thesis-Reference	23	21	2	25%	91%
Total Aktors (4)	**39**	**32**	**7**	**39%**	**82%**
DublinCore:Article	73	9	64	11%	12%
DublinCore:Conference	17	1	16	1%	5%
DublinCore:EditedBook	5	0	5	0%	0%
DublinCore:Journal	23	0	23	0%	0%
DublinCore:Manuscript	9	1	8	1%	11%
DublinCore:Periodical	53	6	47	7%	11%
DublinCore:Thesis	8	1	7	1%	12%
DublinCore:ThesisDegree	20	1	19	1%	5%
Total Dublin Core (5)	**90**	**15**	**75**	**18%**	**17%**
Total Publications (6)	**123**	**46**	**77**	**55%**	**37%**
Generic Ontologies					
Dbpedia:Bibliographic_database	7	0	7	0%	0%
Dbpedia:AcademicJournal	7	1	6	1%	14%
Total DBPedia	**9**	**1**	**8**	**1%**	**11%**
Schema:EducationEvent	3	0	3	0%	0%
Schema:PublicationIssue	0	0	0	0%	0%
Schema:PublicationVolume	0	0	0	0%	0%
Schema:ScholarlyArticle	3	0	3	0%	0%
Total Schema.org	**3**	**0**	**3**	**0%**	**0%**
Total Generic Ontologies	**10**	**1**	**9**	**1%**	**1%**
Total (7)	**123**	**46**	**77**	**55%**	**37%**

(1) Number of triplesets that used the term, out of the 83 triplesets reachable at the time of the experiment and manually classified in (Bizer et al., 2014) in the publications domain.
(2) Percentage of the number of triplesets that used the term over the 83 triplesets reachable at the time of the experiment and manually classified in (Bizer et al., 2014) in the publications domain.
(3) Percentage of the number of domain triplesets over the sum of all triplesets found.
(4) Total number of triplesets that used any of the Aktors terms listed above.
(5) Total number of triplesets that used any of the Dublin core terms listed above.
(6) Total number of triplesets that used any of the Aktors or Dublin core terms listed above.
(7) Total number of triplesets that used any of the Aktors, Dublin core, Schema.org or DBPedia terms listed above.

Table 30. Availability of triplesets classified in the publications domain

name	SPARQL	DUMP	Reachable
agris	Y	N	N
agrovoc-skos	Y	N	N
amsterdam-museum-as-edm-lod	Y	Y	Y
archiveshub-linkeddata	Y	Y	Y
asjp	N	Y	Y
bibbase	Y	N	N
bible-ontology	Y	Y	N
bluk-bnb	Y	N	Y
british-museum-collection	Y	N	N
calames	N	N	N
core	Y	Y	Y
data-bnf-fr	N	N	N
data-open-ac-uk	Y	N	N
datos-bne-es	Y	N	Y
dcs-sheffield	N	Y	Y
deutsche-biographie	Y	N	Y
dewey_decimal_classification	Y	N	N
doi	N	N	N
dspace	N	N	N
dutch-ships-and-sailors	Y	N	Y
ecco-tcp-linked-data	Y	N	N
ecs	N	Y	Y
eur-lex-rdf	Y	Y	N
europeana-lod-v1	Y	Y	N
fu-berlin-dblp	Y	N	N
fu-berlin-project-gutenberg	Y	N	Y
gesis-thesoz	Y	Y	Y
glottolog	N	N	N
hebis-bibliographic-resources	Y	Y	Y
hedatuz	Y	Y	Y
http-www-iwmi-cgiar-org-publications-iwmi-working-papers	N	N	N
hungarian-national-library-catalog	Y	N	Y
idreffr	N	N	N
isidore	Y	N	Y
italian-public-schools-linkedopendata-it	Y	Y	N
iwmi-research-reports	N	N	N
j-ucs-journal-of-universal-computer-science	N	N	N
jiscopenbib-bl_bnb-1	Y	Y	Y
kaken	N	N	N
l3s-dblp	Y	N	Y

lcsh	N	Y	Y
libris	Y	N	N
libver	Y	Y	Y
linkedlccn	Y	N	N
lista-encabezamientos-materia	Y	N	Y
lobid-organisations	N	N	N
lobid-resources	N	N	N
manchester-university-reading-lists	N	N	N
marc-codes	N	N	N
mesh-finnish	N	N	N
morelab	Y	N	Y
msc	Y	Y	Y
multimedia-lab	N	N	N
nalt	N	N	N
nottingham-trent-university-resource-lists	N	N	N
npg	Y	N	Y
ntnusc	N	N	N
nvd	Y	Y	Y
nytimes-linked-open-data	N	Y	Y
oclc-fast	N	N	N
open-library	N	N	N
printed-book-auction-catalogues	Y	N	N
psh-subject-headings	N	Y	Y
radatana	Y	N	Y
rdf-book-mashup	N	N	N
rkb-explorer-acm	Y	Y	Y
rkb-explorer-budapest	Y	Y	Y
rkb-explorer-citeseer	Y	Y	Y
rkb-explorer-courseware	Y	Y	Y
rkb-explorer-crm	Y	N	Y
rkb-explorer-curriculum	Y	Y	Y
rkb-explorer-darmstadt	Y	Y	Y
rkb-explorer-dblp	Y	Y	Y
rkb-explorer-deepblue	Y	Y	Y
rkb-explorer-deploy	Y	Y	Y
rkb-explorer-dotac	Y	Y	Y
rkb-explorer-eprints	Y	Y	Y
rkb-explorer-epsrc	Y	Y	Y
rkb-explorer-era	Y	Y	Y
rkb-explorer-eurecom	Y	Y	Y
rkb-explorer-ft	Y	Y	Y
rkb-explorer-ibm	Y	Y	Y
rkb-explorer-ieee	Y	Y	Y
rkb-explorer-irit	Y	Y	Y
rkb-explorer-italy	Y	Y	Y

rkb-explorer-jisc	Y	Y	Y
rkb-explorer-kaunas	Y	Y	Y
rkb-explorer-kisti	Y	Y	Y
rkb-explorer-laas	Y	Y	Y
rkb-explorer-lisbon	Y	Y	Y
rkb-explorer-newcastle	Y	Y	Y
rkb-explorer-nsf	Y	Y	Y
rkb-explorer-oai	Y	Y	Y
rkb-explorer-os	Y	Y	Y
rkb-explorer-pisa	Y	Y	Y
rkb-explorer-rae2001	Y	Y	Y
rkb-explorer-resex	Y	Y	Y
rkb-explorer-risks	Y	Y	Y
rkb-explorer-roma	Y	Y	Y
rkb-explorer-southampton	Y	Y	Y
rkb-explorer-ulm	Y	Y	Y
rkb-explorer-unlocode	Y	Y	Y
rkb-explorer-wiki	Y	Y	Y
scholarometer	N	Y	Y
semantic-library	N	N	N
semantic-universe	Y	N	N
semantic-web-dog-food	Y	N	N
southampton-ecs-eprints	N	Y	N
st-andrews-resource-lists	N	N	N
stitch-rameau	N	Y	Y
stw-thesaurus-for-economics	Y	Y	Y
sudocfr	Y	N	Y
swedish-open-cultural-heritage	N	Y	Y
sztaki-lod	Y	Y	Y
t4gm-info	N	Y	N
the-european-library-open-dataset	N	Y	N
thesaurus-datenwissen	N	N	N
thesaurus-w	Y	Y	N
thesesfr	Y	N	N
ub-mannheim-linked-data	N	N	N
university-plymouth-reading-lists	N	N	N
university-sussex-reading-lists	N	N	N
verrijktkoninkrijk	Y	N	Y
viaf	N	N	N
vivo-cornell-university	N	Y	Y
vivo-cu-boulder	N	Y	Y
vivo-indiana-university	N	Y	Y
vivo-ponce	N	Y	Y
vivo-scripps-research-institute	N	Y	Y
vivo-university-of-florida	N	Y	Y

vivo-weill-cornell-medical-college	N	Y	Y
vivo-wustl	N	N	N
ysa	N	Y	Y
zbw-labs	N	N	N
zbw-pressemappe20	N	N	N
Total:	**83**	**74**	**83**

7.7. A Behavior Evaluation of the Crawled Resources at Each Level

In this section, we evaluate how the variation of the number of levels affects the final result. Figure 22 to Figure 36 shows, for a selected number of resources from the publication domain, the number of resources found at each level until the 10th level. As expected, all evaluations starts with one resource at the first level, since only one resource was processed for each evaluation. In addition, the graphs presented uses an average of at least three tasks for each initial resource. Furthermore, it is important to remember that the tool does not allows cyclic references.

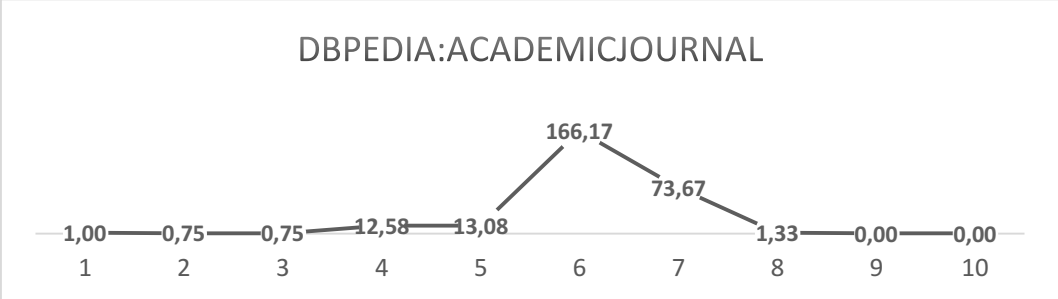

Figure 22. DBPedia:AcademicJournal average

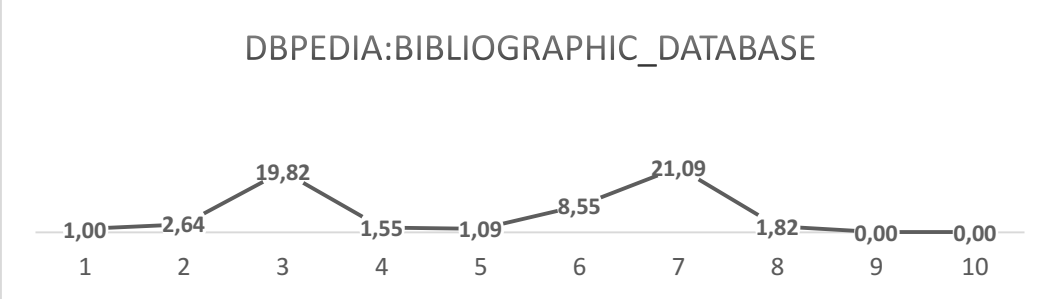

Figure 23. DBPedia:Bibliogragic_database average

Based on the resources crawled, the *reachability* of the DBPedia ontology, that is, after 8 levels the crawler does not find any new resources. In fact, at levels

6 or 7, the crawler was able to found the majority of the resources. The graph for *AcademicJournal*' shows results which are less than 1, which indicates that not every crawling tasks was able to crawl the third level, and so on.

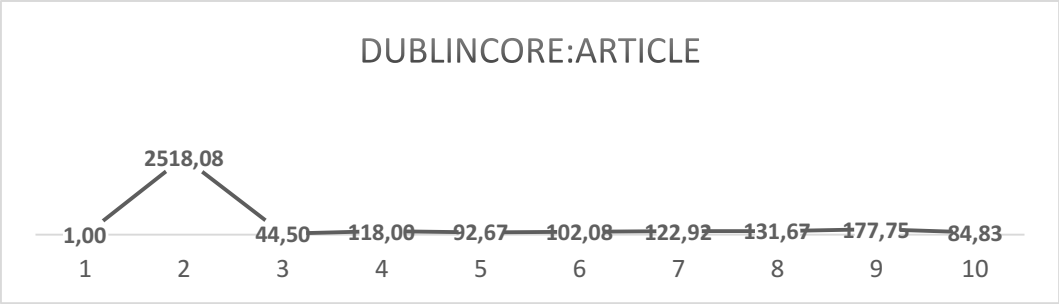

Figure 24. DublinCore:Article average

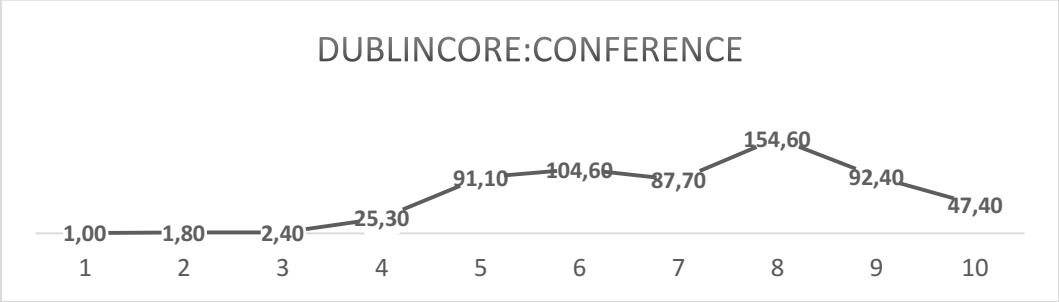

Figure 25. DublinCore:Conference average

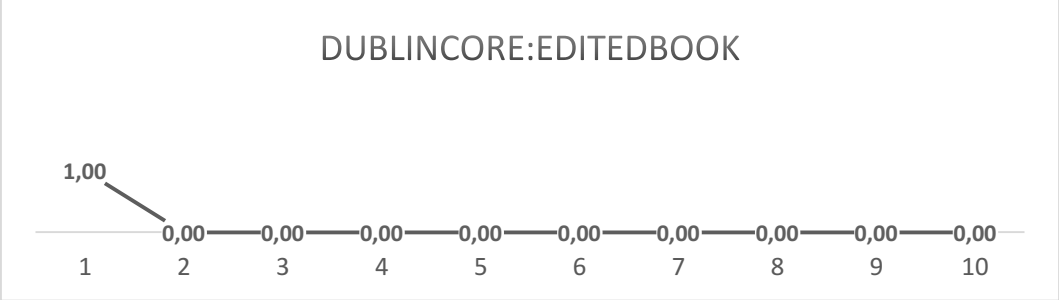

Figure 26. DublinCore:EditedBook average

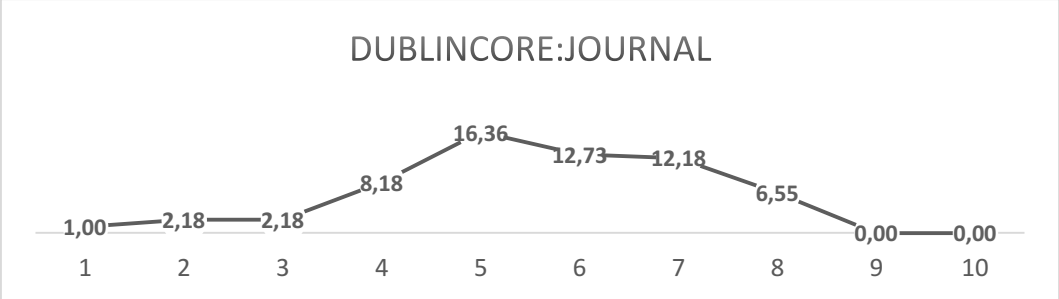

Figure 27. DublinCore:Journal average

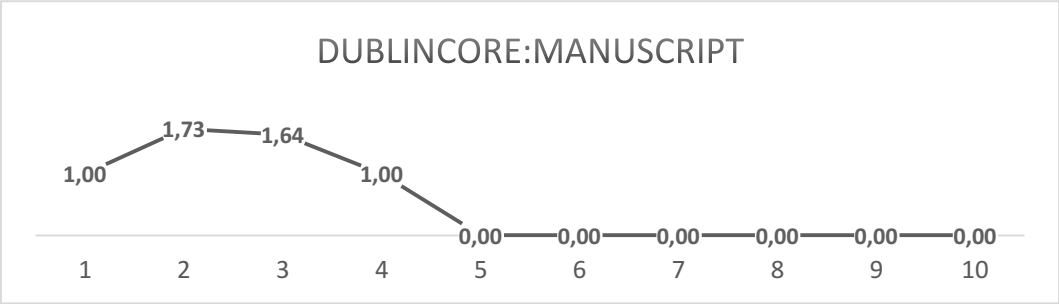

Figure 28. DublinCore:Manuscript average

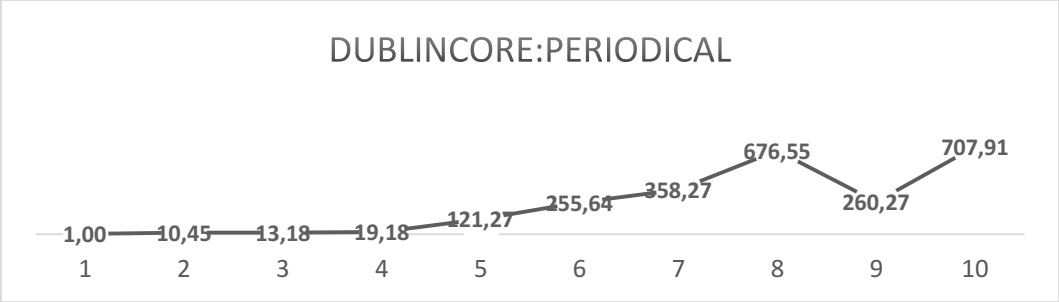

Figure 29. DublinCore:Periodical average

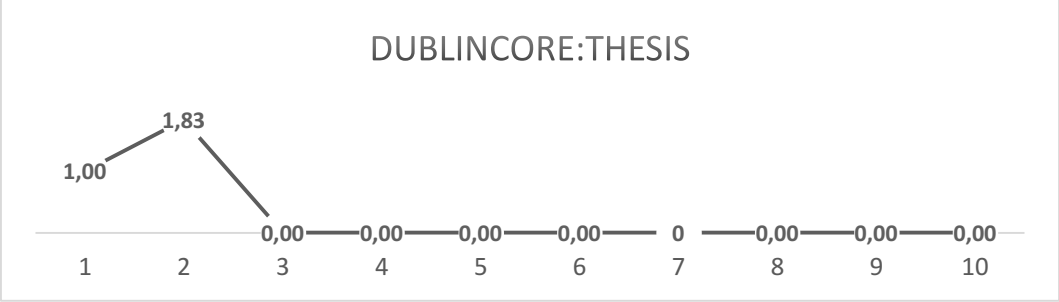

Figure 30. DublinCore:Thesis

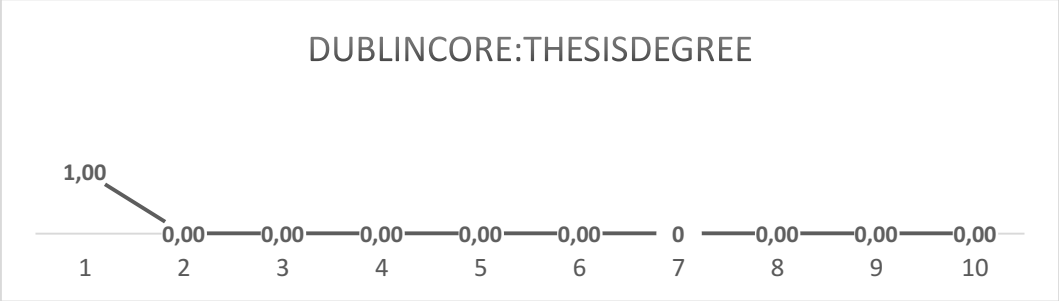

Figure 31. DublinCore:ThesisDegree

The DublinCore ontology presents the most linear series of the three ontologies. For several resources (Figure 24, Figure 25, and Figure 29), the number of resources increases on as the level increases. This kind of behavior may denote that: (1) the DublinCore ontology has many internal relationships that

feed the crawler with more resources, which is unlikely due to the number of resources found; (2) the ontology is commonly accepted by database administrator so that new ontologies are created using relationships using DublinCore to enhance the visibility and knowledge sharing. Those graphs illustrate how a Linked Data resource should behave.

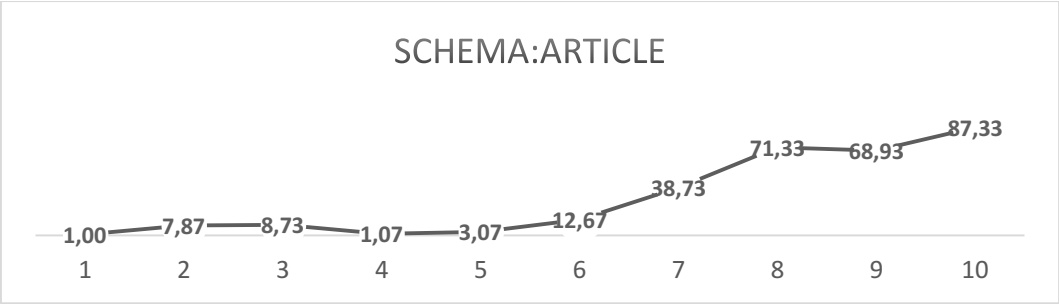

Figure 32. Schema:Article average

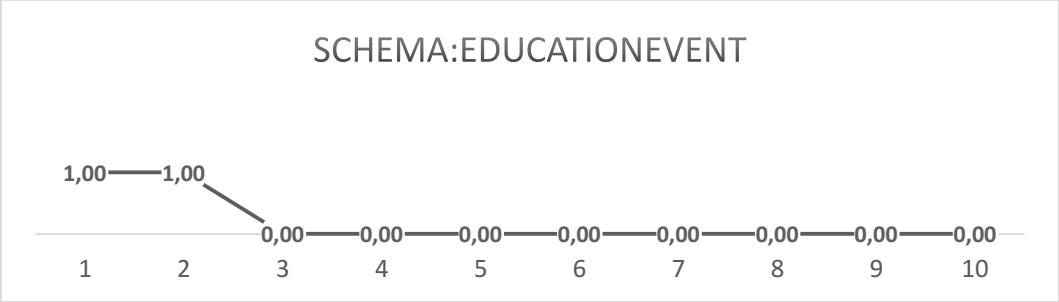

Figure 33. Schema:EducationEvent average

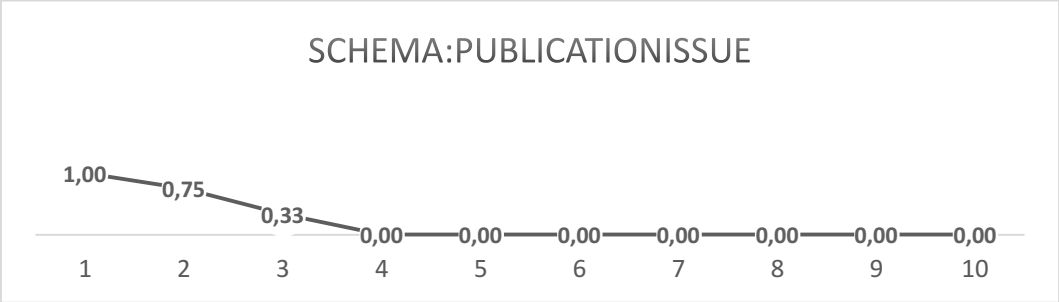

Figure 34. Schema:PublicationIssue average

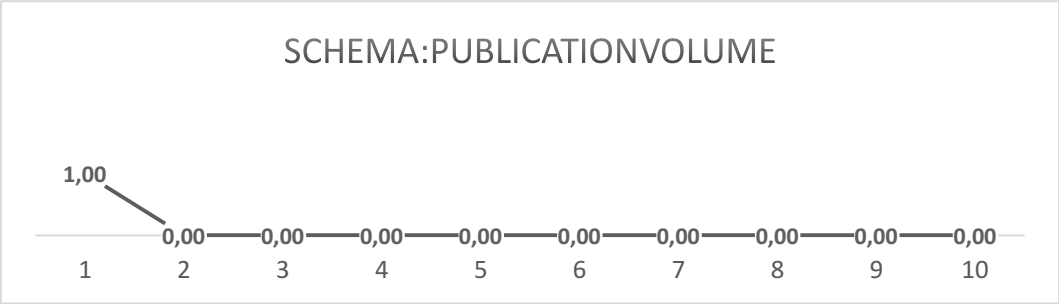

Figure 35. Schema:PublicationVolume average

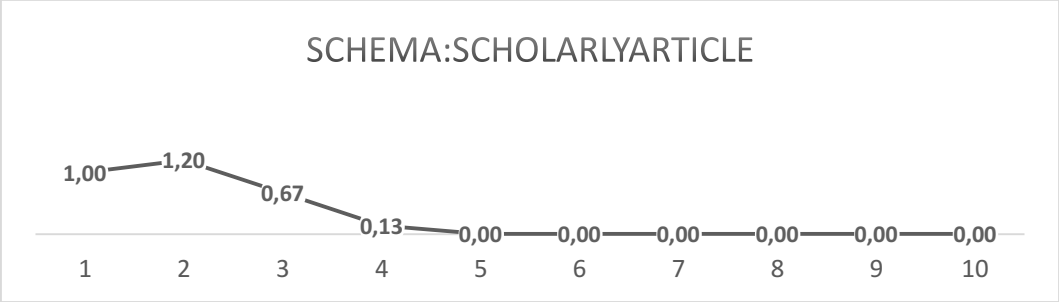

Figure 36. Schema:ScholarlyArticle average

The graphs for the Schema.org ontology shows that it currently is a useful ontology, providing generic definitions. Article, which was the only resource to have a significant number of related resources found, is a generic term that can be used to describe a Journal text, a Blog Post, a conference paper and others.

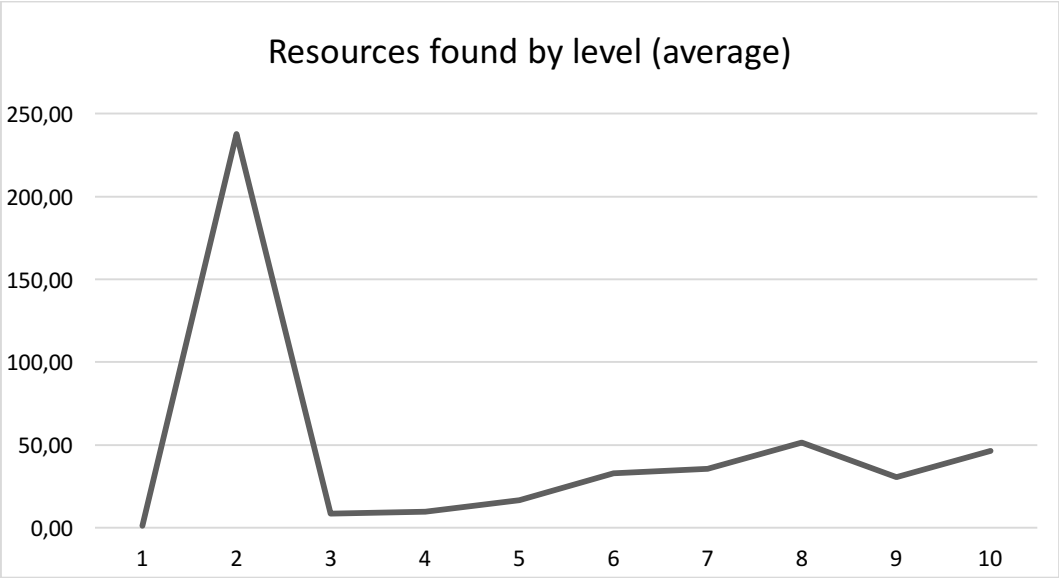

Figure 37. All resources average

Figure 37 shows the average number of terms found by all resources crawled. The graph shows a growing linear series. In other words, it is desirable to

crawl as many levels as possible since the crawler is able to reach more resources at each new level. With processing time restrictions, we recommend that the crawling task must reach at least the 5th level since it is a balance between the number of resources found (more then half of the results will be found) and the processing time.

8.
Conclusions and Suggestions for Future Work

8.1.
Conclusions

In this thesis, we concentrated on the development of a metadata-focused crawler for the Linked Open Data (LOD) cloud that could be used to crawl and identify resources (triples and ontologies) that are related to a specific subject. We addressed the challenges highlighted in Section 1.1 that are related to the difficulty of finding related open triplesets, as the LOD cloud grows.

We may highlight the following contributions of this thesis to the area of Linked Data crawling:

Crawling Strategy

- *Crawling with SPARQL queries*. Our crawler returns richer metadata than a traditional crawler, since they use SPARQL queries, executed over all triplesets. In particular, our crawler discovers not only the links between resources, but also the number of instances related to the crawling terms.
- *Identifying resources in different languages and alphabets*. Our crawler was able to identify resources in different languages, even in different alphabets, through the *sameAs* and *seeAlso* queries.
- *Leveraging grouping functions in SPARQL queries*. Grouping functions were introduced in SPARQL 1.1 and helped us optimize some queries and reduce performance problems.
- *Discovering relationships between resources of two triplesets described in a third one*. While using our crawler, we found cases in which a relationship between two resources *r* and *r'*, respectively, defined in triplesets *d* and *d'*, were described in another tripleset *d"*. This happens, for example, when the ontologies used by *d* and *d'* are only stored in a different dataset *d"*. In these cases, it was necessary to crawl triplesets, other than *d* and *d'*, to find the

relationship between *r* and *r'*. A traditional crawler following links from *d* (or *d'*) would not find any link between *r* and *r'* because it is only declared in *d''*.

- *Performing simple deductions*. When using the provenance list generated by the crawler, one may perform simple deductions, using the transitivity of the subclass property, perhaps combined with the *sameAs* relationship. For example, suppose that the crawler discovered that `opencyc:Hit_music` is a subclass of `opencyc:Music`, which in turn has a *sameAs* relationship with `wordnet:synset-music-noun-1`. Then, one may deduce that `opencyc:Hit_music` is a subclass of `wordnet:synset-music-noun-1`.

Crawler Architecture

- *A Crawler Framework*. The intention when implementing CrawlerLD was to create a tool that could be expandable. To achieve this objective, we created the concept of *processors*, which can be implemented by any developer and plugged into their tool. Furthermore, the tool is bundled with three processors that can be enabled or disabled when the user wants to.
- *Applying the actor model to crawling tasks*. Introduced in 1973, the adoption of the actor model to design applications in different domains seems to have recently increased. All such applications share the following characteristics: they need to be distributed, scalable, and use few resources. In this thesis, we presented a solution to create a Linked Data Crawling Framework using the actor model, and we evaluated how it performed in comparison with traditional solutions.
- *Distributed Linked Data Crawler*. The LOD cloud has been growing steadily (Bizer et al., 2014) as its popularity increases. It is not possible to suppose that a single computer will be capable to crawl all LOD cloud in the following years, in a reasonable time. In this thesis, we demonstrate how CrawlerLD can be configured to be used in as many machines as a user wants.

8.2. Suggestions for Future Work

As for future research, we suggest:

Improvements to the Crawler Architecture

- *Create new processors that explore other Linked Data characteristics* - CrawlerLD was designed to be expansible. In other words, any developer can create new processors that will be integrated into the core system, and that will be used in each crawling level. Some suggestions for new processors are:
    - Linked Data Fragments (Verborgh et al., 2014) – Linked Data Fragments are triplesets features similar to a SPARQL Endpoint. Analogous to relational databases, we can compare SPARQL to a SQL Query and a Linked Data Fragment to a database view. Triplesets administrators could optimize these fragments to be more efficient than SPARQL endpoints. The CrawlerLD could have a new processor created only to process resources through these fragments.
    - Other properties – There are a number of properties defined in RDF and OWL that can be used to increase precision and recall. Such properties would be handled by specific processors.
    - VoID – Although previous tests revealed that VoID ontology is not useful to find new data about a resource, it is a good idea to idea create a processor to extract this kind of data. The major advantage will be to follow it, if this behavior will be maintained in the following years.
    - SameAs processor – SameAs.org[30] is library of owl:sameAs relationships. It provides a REST API that can be used by our tool, in order to easily discover more relationships.
- *Create a crawling ontology to model the output of a crawling task* - Currently, the output of a crawling task is a binary file that can be only processed by CrawlerLD (though it can be parsed using the core API or Rest API). This can be changed, so the result could be saved in an RDF file using a specific ontology to represent the mapping (with any provenience) found. Previous research did not find any ontology designed to represent this kind of application.
- *Open source distribution of the Utilities Semantic Web and CrawlerLD* – There was an effort to make the tool and its libraries easy to learn and deploy. In addition, we created an API, called *Utilities Semantic Web*, which is capable of identifying new triplesets on the LOD cloud and of querying

[30] http://sameas.org/

SPARQL Endpoints, RDF Dumps, and URI's. These tools could be distributed as open source software.

Improvements to the Crawler Performance

- *Use older crawling data as cache for new ones* – many crawling tasks use resources that, in some way, appear in older tasks. Hence, caching older results may considerably improve new crawling tasks.
- *Use the LOD Laudromatic (Beek et al., 2014)* – The *LOD Laudromatic* takes a snapshot of the LOD cloud and makes it available to anyone. Therefore, we could download and replicate the snapshot to improve performance of the crawler. This would also help compare the performance of the various crawlers, since the LOD cloud is always evolving, which makes it difficult to replicate experiments.

Other Uses to the Crawler

- *Evaluation of popular ontologies* – Since the beginning of this project, we saw a number of popular ontologies disappear and others become more popular. The crawler can be used to measure the popularity of an ontology over the LOD cloud.
- *Create a recommender system based on the output of the crawler* - our vision for the crawler is to use it in a broader environment in which a user actually uses a recommender system to design their tripleset. The recommender system could use the *Core API* or the *Rest API*. An advantage of the use of the Rest Service is that it is better aligned to the Microservices Architecture (Newman, 2015). Furthermore, a load balancer could be applied to handle multiple crawler instances.

Bibliography

Assaf, A., Senart, A., & Troncy, R. (2015). **Roomba: Automatic Validation, Correction and Generation of Dataset Metadata**. In Proceedings of the 24th International Conference on World Wide Web Companion (pp. 159-162). International World Wide Web Conferences Steering Committee.

Alexander, K., & Hausenblas, M. (2009). **Describing linked datasets-on the design and usage of void, the'vocabulary of interlinked datasets**. In In Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09.

Baeza-Yates, Ricardo, and Berthier Ribeiro-Neto. **Modern information retrieval**. Vol. 463. New York: ACM press, 1999.

Beek, W., Rietveld, L., Bazoobandi, H. R., Wielemaker, J., & Schlobach, S. (2014). **LOD laundromat: a uniform way of publishing other people's dirty data**. In The Semantic Web–ISWC 2014 (pp. 213-228). Springer International Publishing.

Berners-Lee, Tim. **Linked data-design issues**. (2006). Available at http://www.w3.org/DesignIssues/LinkedData.html

Bizer, C., Heath, T., & Berners-Lee, T. (2009). **Linked data-the story so far**. Semantic Services, Interoperability and Web Applications: Emerging Concepts, 205-227.

Bizer, C., A. Jentzsch, and R. Cyganiak. (2014). **State of the LOD cloud: Version 0.4**. Available at *http://www4.wiwiss.fu-berlin.de/lodcloud/state*.

Brickley, D., Guha, R.V. (eds.). (2004). **RDF Vocabulary Description Language 1.0: RDF Schema**. *W3C Recommendation* 10 February 2004.

De Bra, P. M., & Post, R. D. J. (1994). **Information retrieval in the World-Wide Web: making client-based searching feasible**. Computer Networks and ISDN Systems, 27(2), 183-192.

Hewitt, C., Bishop, P., & Steiger, R. (1973, August). **A universal modular actor formalism for artificial intelligence**. In Proceedings of the 3rd international joint conference on Artificial intelligence (pp. 235-245). Morgan Kaufmann Publishers Inc..

De Assis, G. T., Laender, A. H., Gonçalves, M. A., & Da Silva, A. S. (2009). **A genre-aware approach to focused crawling**. World Wide Web, 12(3), 285-319.

Ding, L., Pan, R., Finin, T., Joshi, A., Peng, Y., & Kolari, P. (2005). **Finding and ranking knowledge on the semantic web**. In The Semantic Web–ISWC 2005 (pp. 156-170). Springer Berlin Heidelberg.

Fielding, R. T., & Taylor, R. N. (2002). **Principled design of the modern Web architecture**. ACM Transactions on Internet Technology (TOIT), 2(2), 115-150.

Fionda, V., Gutierrez, C., & Pirró, G. (2012, April). **Semantic navigation on the web of data: specification of routes, web fragments and actions**. In Proceedings of the 21st international conference on World Wide Web (pp. 281-290). ACM.

Garlik, Steve H., Andy Seaborne, and Eric Prud'hommeaux. **SPARQL 1.1 query language**. World Wide Web Consortium (2013).

Gomes, Raphael do Vale A.; Casanova, Marco A.; Lopes, Giseli Rabello; Leme, Luiz André P. Paes. (2014). **A Metadata Focused Crawler for Linked Data**. In: The 16th International Conference on Enterprise Information Systems - ICEIS, 2014, Lisbon, Portugal. Proceedings of the 16th International Conference on Enterprise Information Systems. SCITEPRESS - Science and Technology Publications, 2014. v.2. p.489 - 500.

Raphael do Vale, A. G., Casanova, M. A., Lopes, G. R., & Leme, L. A. P. P. (2015). **CRAWLER-LD: A Multilevel Metadata Focused Crawler Framework for Linked Data**. In Enterprise Information Systems (pp. 302-319). Springer International Publishing.

Hartig, O., Bizer, C., & Freytag, J. C. (2009). **Executing SPARQL queries over the web of linked data** (pp. 293-309). Springer Berlin Heidelberg.

Hersovici, M., Jacovi, M., Maarek, Y. S., Pelleg, D., Shtalhaim, M., & Ur, S. (1998). **The shark-search algorithm. An application: tailored Web site mapping**. Computer Networks and ISDN Systems, 30(1), 317-326.

Isele, R., Umbrich, J., Bizer, C., & Harth, A. (2010, November). **LDspider: An open-source crawling framework for the Web of Linked Data**. In 9th International Semantic Web Conference (ISWC2010).

Leme, L. A. P. P., Lopes, G. R., Nunes, B. P., Casanova, M. A., & Dietze, S. (2013). **Identifying candidate datasets for data interlinking**. In Web Engineering (pp. 354-366). Springer Berlin Heidelberg.

Lopes, G. R., Leme, L. A. P. P., Nunes, B. P., Casanova, M. A., & Dietze, S. (2013). **Recommending tripleset interlinking through a social network approach**. In Web Information Systems Engineering–WISE 2013 (pp. 149-161). Springer Berlin Heidelberg.

Manola, F., Miller, E., 2004. **RDF Primer**, *W3C Recommendation* 10 February 2014.

Romero, M. M., Vázquez-Naya, J. M., Munteanu, C. R., Pereira, J., & Pazos, A. (2010). **An approach for the automatic recommendation of ontologies using collaborative knowledge**. In Knowledge-Based and Intelligent Information and Engineering Systems (pp. 74-81). Springer Berlin Heidelberg.

Newman, Sam (2015). ***Building Microservices***. New York: O'Reilly Media. 280.

Nikolov, A., d'Aquin, M. (2011). **Identifying Relevant Sources for Data Linking using a Semantic Web Index**. In Proc. Workshop on Linked Data on the Web. CEUR-WS.org.

Nikolov, A., d'Aquin, M., & Motta, E. (2012). **What should I link to? Identifying relevant sources and classes for data linking**. In The Semantic Web (pp. 284-299). Springer Berlin Heidelberg.

Prud'hommeaux, E., Seaborne, A., 2008. **SPARQL Query Language for RDF**, *W3C Recommendation* 15 January 2009.

Saint-Paul, R., Raschia, G., & Mouaddib, N. (2005, August). **General purpose database summarization**. In Proceedings of the 31st international conference on Very large data bases (pp. 733-744). VLDB Endowment.

Srinivasan, P., Menczer, F., & Pant, G. (2005). **A general evaluation framework for topical crawlers**. Information Retrieval, 8(3), 417-447.

Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., & Van de Walle, R. (2014, April). **Web-scale querying through linked data fragments**. In Proceedings of the 7th Workshop on Linked Data on the Web.

Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., ... & Van de Walle, R. (2014). **Querying datasets on the web with high availability**. In The Semantic Web–ISWC 2014 (pp. 180-196). Springer International Publishing.

W3C OWL Working Group, 2012. **OWL 2 Web Ontology Language Document Overview (Second Edition)**. *W3C Recommendation* 11 December 2012.

Wang, J., Wen, J. R., Lochovsky, F., & Ma, W. Y. (2004, August). **Instance-based schema matching for web databases by domain-specific query probing**. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30 (pp. 408-419). VLDB Endowment.

Annex A – Pseudo-code of the Basic Implementation of Chapter 4

genericCrawlingQuery(*d, S, t, p; R*);

input: *d* - direction of the query (“direct” or “reverse”)
S - a SPARQL Endpoint or a RDF Dump to be queried
t - a crawling term
p - a predicate
output: *R* - a set of terms crawled from *t*

begin
if d == “direct”
then R := **execute** SELECT distinct ?item WHERE { ?item *p* <*t*> } **over** *S*
else R := **execute** SELECT distinct ?item WHERE { <*t*> *p* ?item } **over** *S*
return *R*;
end

```
CRAWLER(maxLevels, maxTerms, maxFromTerm, maxFromSet; T, C; Q, P, D)

Parameters: maxLevels     - maximum number of levels of the breadth-first search
            maxTerms      - maximum number of terms probed
            maxFromTerm   - maximum number of new terms probed from each term
            maxFromSet    - maximum number of terms probed from a tripleset, for each term
input:      T   - a set of input terms
            C   - a list of catalogues of triplesets
output:     Q   - a queue with the terms that were crawled
            P   - a provenance list for the terms in Q
            D   - a provenance list of the triplesets with terms in Q

begin   Q, P, D := empty;
        #levels, #terms := 0;
        nextLevel := T;
        while #levels < maxLevels and #terms < maxTerms do
        begin
            #levels := #levels + 1;
            currentLevel := nextLevel;      /* currentLevel and nextLevel are queues of terms */
            nextLevel := empty;
            for each t from currentLevel do
            begin
                    add t to Q;
                    /*                      crawling by dereferencing                      */
                    S := downloaded RDF content obtained by dereferencing t;
                    R1 := empty;
                    for each predicate p in { rdfs:subClassOf,owl:sameAs,rdfs:seeAlso } do
                    begin
                    if p == "rdfs:subClassOf" then d := "direct" else d := "inverse";
                    genericCrawlingQuery( d, S, t, p; RTEMP );
                    if (RTEMP not empty)
                            then  begin add (t, p, RTEMP, S) to P;
                                        R1 := concatenate(R1, RTEMP);
                                  end
                    end
                    /*                      crawling by direct querying the triplesets in C        */
                    R2 := empty;
                    for each tripleset S from the catalogues in C do
                    begin
                    RS := empty;
                    for each predicate p in { rdfs:subClassOf,owl:sameAs,rdfs:seeAlso } do
                    begin
                                genericCrawlingQuery( "direct", S, t, p; RTEMP );
                                if (RTEMP not empty)
                                then  begin add (t, p, RTEMP, S) to P;
                                            RS := concatenate(RS, RTEMP);
                                      end
                    end
                    if (RS not empty)
                            then  begin add (t, S) to D;
                                        truncate RS to contain just the first maxFromSet terms;
                                        R2 := concatenate(R2, RS);
                                  end
                    end
                    RT := concatenate(R1, R2)
                    for each u in RT do
                    begin
                    #termsFromTerm := #termsFromTerm +1;
                    #terms := #terms +1;
                                if ( #termsFromTerm > maxFromTerm or #terms > maxTerms ) then exit;
                                add u to nextLevel;
                     end
                    end
                    end
          return Q, P, D;
        end
```

Annex B – Pseudo-code of CrawlerLD and DIST-CrawlerLD

```
CRAWLER-LD(maxLevels, maxTerms, maxFromTerm, maxFromSet; T, C, PR; Q, P, D)

Parameters: maxLevels     - maximum number of levels of the breadth-first search
            maxTerms      - maximum number of terms probed
            maxFromTerm   - maximum number of new terms probed from each term
            maxFromSet    - maximum number of terms probed from a tripleset, for each term
input:      T   - a set of input terms
            C   - a list of catalogues of triplesets
            PR  - a list of processors
output:     Q   - a queue with the terms that were crawled
            P   - a provenance list for the terms in Q
            D   - a provenance list of the triplesets with terms in Q

begin   Q, P, D := empty;
        #levels, #terms := 0;
        nextLevel := T;
        while #levels < maxLevels and #terms < maxTerms do
        begin
            #levels := #levels + 1;
            currentLevel := nextLevel;      /* currentLevel and nextLevel are queues of terms */
            nextLevel := empty;
            for each t from currentLevel do
            begin

                    terms += terms;
                    if ( #terms > maxTerms ) then exit;
                    add t to Q;
                    resourcesForEachDataset := (dataset,resourceList) := empty
                    for each p from PR do
                    begin
                    /* use t on the processor p and save the results for each dataset */
                     call (dataset,resultList) := p(t,P,D)
                     add p to resourceForEachDataset
                    end
                    /* limiting results phase */
                    resourcesFromTerm := empty
                    for each dataset d from resourcesForEachDataset
                    begin
                     resultList := results from dataset D on term t;
                     truncate resultList to contain just the first maxFromSet terms;
                     resourcesFromTerm := concatenate(resultList, resourcesFromTerm);
                    end
                    truncate resourcesFromTerm to contain just the first maxFromTerm terms;
                     nextLevel := concatenate(resourcesFromTerm, nextLevel);

            end /* t loop */

        end /* level loop */

        return Q, P, D;

end /* algorithm */
```

Annex C – A Brief Tutorial to Create a Processor in DIST-CrawlerLD

DIST-CrawlerLD and its previous versions (Chapters 7 and 6, respectively) have the ability to receive new custom-made processors. This annex will briefly describe how to create a new processor in DIST-CrawlerLD.

A developer needs to take two steps to create a new processor on the tool: (1) create a new class extending `ProcessorActor` class; and (2) register the newly created class in `ProcessorManager`.

Creating a new processor

`ProcessorActor` is a class that saves all parameters specified by a user and allows the developer to create any type of processor. It will not force the developer to use a specific resource or anything related. The developer just has to write code to handle the message *Calculate* (Figure 38).

```
@Override
public void onReceive(Object message) throws Exception {
    if(message instanceof Calculate){
        calculate();
    }
    else if(message instanceof SparqlResultset){
        processQueryResult((SparqlResultset) message);
    }
    else if(message instanceof QueryFinishedMessage){
        finishQueryResult((QueryFinishedMessage)message);
    }
    else{
        unhandled(message);
    }
}
```

Figure 38. Handling calculation messages.

If the developer wants to create a processor that makes queries in all datasets in a distributed (if configured) way, they may simply extend the class `AbstractQuerierProcessor`. All bundled processors extend this class, which is capable of querying the LOD Cloud and returning the result in a simple way. To extend this class, the user will need to implement the following methods:

- calculate – `AbstractQuerierProcessor` will tell when the processor needs to start its processing. It can use the method `sendQuery(Query, Identifier)`, `sendQuery(Query,Identifier,Dataset)`, or `sendQuery(Query, Identifier, List<Dataset>)` to send queries over datasets, using what was shown in Section 7.3.
- processQueryResult – receives the result of a single tripleset and allows the processor to handle the resultset appropriately.
- finishQueryResult – indicated that a query sent by the processor (identified by `identifier` parameter) finished the processing on all datasets.

Registering the processor.

Currently, this step is hardcoded and a developer needs to register by changing the source code of `ProcessorManager` class. Figure 39 illustrates how a developer should register the processor.

```
public static Set<Class<? extends ProcessorActor>> getProcessors()
{
    List<Class<? extends ProcessorActor>> notValidatedProcessorList = Arrays.asList(
                DereferenceProcessor.class
            ,   NumberOfInstancesProcessor.class
            ,   PropertyQueryProcessor.class
    );

    Set<Class<? extends ProcessorActor>> processorList = new HashSet<>();

    for (Class<? extends ProcessorActor> processor : notValidatedProcessorList) {
        if(validateConstructor(processor)){
            processorList.add(processor);
        }
        else{
            String msg = String.format("Processor %s not valid because it does not ha
                    "Will be ignored.", processor);
            throw new RuntimeException(msg);
              LOG.warn(msg);
        }
    }

    return processorList;
}
```

Figure 39. Registering a processor

Future versions of DIST-CrawlerLD will address how to simplify this process by removing mandatory registration at *ProcessorManager* class.