FACULTY OF ENGINEERING AND ARCHITECTURE

Discovering Digital Art Collections using Link-Traversal-based Query Processing

Martijn Bogaert
Student number: 01706456

Supervisors: Prof. dr. Pieter Colpaert, Prof. dr. ir. Ruben Verborgh
Counsellors: Bryan-Elliott Tam, Ir. Wout Slabbinck

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2022-2023

GHENT UNIVERSITY

Acknowledgements

TODO

Notes related to the master's thesis

This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text.

AI usage

TODO

Abstract

TODO

Conference Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

This document is a model and instructions for LaTeX. Please observe the conference page limits.

II. Ease of Use

A. Maintaining the Integrity of the Specifications

The IEEEtran class file is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

III. Prepare Your Paper Before Styling

Before you begin to format your paper, first write and save the content as a separate text file. Complete all content and organizational editing before formatting. Please note sections III-A–III-E below for more information on proofreading, spelling and grammar.

Keep your text and graphic files separate until after the text has been formatted and styled. Do not number text heads—LaTeX will do that for you.

Identify applicable funding agency here. If none, delete this.

A. Abbreviations and Acronyms

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, ac, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

B. Units

- Use either SI (MKS) or CGS as primary units. (SI units are encouraged.) English units may be used as secondary units (in parentheses). An exception would be the use of English units as identifiers in trade, such as “3.5-inch disk drive”.
- Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.
- Do not mix complete spellings and abbreviations of units: “Wb/m^2” or “webers per square meter”, not “webers/m^2”. Spell out units when they appear in text: “. . . a few henries”, not “. . . a few H”.
- Use a zero before decimal points: “0.25”, not “.25”. Use “cm^3”, not “cc”.)

C. Equations

Number equations consecutively. To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use a long dash rather than a hyphen for a minus sign. Punctuate equations with commas or periods when they are part of a sentence, as in:

$$a + b = \gamma \tag{1}$$

Be sure that the symbols in your equation have been defined before or immediately following the equation. Use “(1)”, not

"Eq. (1)" or "equation (1)", except at the beginning of a sentence: "Equation (1) is . . ."

D. LaTeX-Specific Advice

Please use "soft" (e.g., `\eqref{Eq}`) cross references instead of "hard" references (e.g., `(1)`). That will make it possible to combine sections, add equations, or change the order of figures or citations without having to go through the file line by line.

Please don't use the `{eqnarray}` equation environment. Use `{align}` or `{IEEEeqnarray}` instead. The `{eqnarray}` environment leaves unsightly spaces around relation symbols.

Please note that the `{subequations}` environment in LaTeX will increment the main equation counter even when there are no equation numbers displayed. If you forget that, you might write an article in which the equation numbers skip from (17) to (20), causing the copy editors to wonder if you've discovered a new method of counting.

BibTeX does not work by magic. It doesn't get the bibliographic data from thin air but from .bib files. If you use BibTeX to produce a bibliography you must send the .bib files.

LaTeX can't read your mind. If you assign the same label to a subsubsection and a table, you might find that Table I has been cross referenced as Table IV-B3.

LaTeX does not have precognitive abilities. If you put a `\label` command before the command that updates the counter it's supposed to be using, the label will pick up the last counter to be cross referenced instead. In particular, a `\label` command should not go before the caption of a figure or a table.

Do not use `\nonumber` inside the `{array}` environment. It will not stop equation numbers inside `{array}` (there won't be any anyway) and it might stop a wanted equation number in the surrounding equation.

E. Some Common Mistakes

- The word "data" is plural, not singular.
- The subscript for the permeability of vacuum μ_0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter "o".
- In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
- A graph within a graph is an "inset", not an "insert". The word alternatively is preferred to the word "alternately" (unless you really mean something that alternates).
- Do not use the word "essentially" to mean "approximately" or "effectively".
- In your paper title, if the words "that uses" can accurately replace the word "using", capitalize the "u"; if not, keep using lower-cased.
- Be aware of the different meanings of the homophones "affect" and "effect", "complement" and "compliment", "discreet" and "discrete", "principal" and "principle".
- Do not confuse "imply" and "infer".
- The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the "et" in the Latin abbreviation "et al.".
- The abbreviation "i.e." means "that is", and the abbreviation "e.g." means "for example".

An excellent style manual for science writers is [7].

F. Authors and Affiliations

The class file is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

G. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is "Heading 5". Use "figure caption" for your Figure captions, and "table head" for your table title. Run-in heads, such as "Abstract", will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced.

H. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. 1", even at the beginning of a sentence.

TABLE I
TABLE TYPE STYLES

Table Head	Table Column Head		
	Table column subhead	*Subhead*	*Subhead*
copy	More table copy[a]		

[a]Sample of a Table footnote.

Fig. 1. Example of a figure caption.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
[4] K. Elissa, "Title of paper if known," unpublished.
[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.
[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

Contents

List of Figures

List of Tables

List of Code Fragments

Introduction

Digital art collections have long stood as a testament to human creativity and cultural evolution. With the advent of technology, many of these collections have undergone digitization, making them more accessible to a global audience. This digitization not only preserves the integrity of the artworks but also offers an opportunity for deeper exploration and understanding. However, with this digital transformation comes a set of challenges, especially for those without a technical background. Professionals in the cultural domain and general art enthusiasts, while passionate about art, may not possess the technical expertise to navigate and query these digitized datasets. This limitation can hinder their ability to make new discoveries and truly immerse themselves in the digital art world.

Discovering art collections can be interpreted in myriad ways. At its core, discovery is about unearthing new insights, understanding the nuances of each artwork, and drawing connections that might not be immediately apparent. This research primarily focuses on retrieving the inherent properties of cultural objects, delving into the intricate details that make each piece unique. However, the true potential of discovery lies in going beyond the confines of a single dataset. Link traversal offers this opportunity, allowing for a broader exploration that extends beyond the immediate dataset, unveiling new layers of knowledge and understanding.

By employing link traversal, one can uncover hidden relationships, gain a deeper understanding of cultural objects, and even compare different artworks in novel and enlightening ways. This approach is particularly beneficial when exploring the Collections of Ghent (CoGhent), a collaborative initiative between various cultural institutions. Published in a Linked Data format, the CoGhent collections are primed for link traversal, enabling a richer and more comprehensive exploration.

This research situates itself at the intersection of art and technology, aiming to bridge the gap between the two. It seeks to empower both professionals and art enthusiasts to navigate the digital art landscape, harnessing the power of link traversal to make new discoveries and draw meaningful connections. Through a systematic exploration of the Collections of Ghent and the development of tools tailored for query formulation, this research offers a roadmap for discovering digital art collections in their entirety.

Chapter 1 elucidates the foundational concepts of Linked Data and their real-world applications. It delves into the core principles, data modeling, and various RDF syntaxes, setting the stage for a deeper exploration of link traversal in the subsequent chapters.

Chapter 2 focuses on the CoGhent collections, highlighting the potential of link traversal for discovering properties of Human-Made Objects. It provides an overview of the available data sources and the development of a link traversal engine optimized for the objectives of this research.

In Chapter 3, the emphasis shifts to the development of user-centric tools for query formulation. Two conceptual web applications are introduced, designed to alleviate the technical complexities of query formulation for users. The chapter also discusses the fundamental functionality shared by both web applications, ensuring a cohesive exploration throughout.

Lastly, Chapter 4 addresses the challenges of visualizing and preserving query results. It offers an overview of potential solutions, outlining their advantages and drawbacks, ensuring that the treasures within the CoGhent collections are accessible and meaningful to all.

1

Related Work

The realm of Linked Data, particularly in the context of digital art collections, has witnessed significant advancements. This chapter seeks to elucidate the foundational concepts and their real-world applications.

Section 1.1 provides an introduction to Linked Data, emphasizing its core principles, data modeling, and various RDF syntaxes. The section underscores the importance of unique URIs, dereferencing, and data interlinking.

In Section 1.2, the spotlight is on Link-Traversal-based Query Processing (LTQP). Through an example, the intricacies of querying across different documents are unraveled, highlighting the challenges and the specific reachability criteria for link traversal.

Section 1.3 delves into Comunica, a SPARQL query engine. The discussion revolves around its modularity, the foundational building blocks, and the potential to craft custom engine configurations tailored for distinct link traversal requirements.

Section 1.4 presents the Collections of Ghent (CoGhent) initiative, a collaborative venture between cultural institutions in Ghent. The adoption of Linked Data Event Streams (LDES) for publishing digital collections is explored, alongside the CoGhent Query Builder application that aids in query formulation.

Concluding the chapter, Section 1.5 introduces the International Image Interoperability Framework (IIIF). Namely, the role of IIIF Manifests and IIIF Viewers in the visualization of cultural data is discussed.

These sections provide the foundation for the subsequent chapters, which delve into various stages of a systematic process for discovering digital art collections. Each chapter builds upon the insights and methodologies presented in this chapter, ensuring a cohesive exploration throughout.

1.1 Linked Data

This section presents a comprehensive exploration of Linked Data, encompassing its fundamental principles, data modeling, syntax, query interfaces, and the associated challenges and advantages. In Section 1.1.1, the concept of Linked Data and its principles are introduced, highlighting the significance of unique URIs, dereferencing, and data interlinking. Section 1.1.2 focuses on the Resource Description Framework (RDF) as the cornerstone for representing relationships and knowledge connections within Linked Data. Section 1.1.3 provides an overview of RDF syntax, including popular formats such as XML,

Turtle, N-Triples, and JSON-LD, which facilitate the flexible expression and exchange of RDF data. Lastly, Section 1.1.4 briefly introduces SPARQL, the query language for RDF data This comprehensive examination serves as a solid foundation for the subsequent discussions on Linked Traversal-based Query Processing.

1.1.1 Introduction and Principles

To better understand the origins of the idea behind Linked Data, it is important to examine the origins of the World Wide Web. For example, its first, but still rather primitive, underlying technology was introduced in 1989 at CERN. Tim Berners-Lee was the man responsible for its development. By using HyperText Markup Language (HTML), it enabled scientists, and later the rest of the world, to publish documents that could contain links to other documents. This helped create a mesh of documents and information. However, since these documents in fact contained nothing more than raw data dumps and links between documents represented simply an indication of how to reach the document, these documents and their relationships lacked semantics. Figure 1.1 illustrates what a web of documents without unambiguous indications of what their contents and the links between them represent, might look like. It is necessary to note here that the used icons are not the contents of their respective documents, but only a representation of their contents. Nevertheless, in themselves, they prove the weakness of such web as much as when the effective content of the documents had been represented. After all, just from the raw content of documents and their mutual links, a person cannot clearly infer exactly what their constellation represents, let alone a computer. From that deficiency, therefore, emerged the idea of Linked Data. (Jacksi and Abass, 2019) (Bizer et al., 2011)

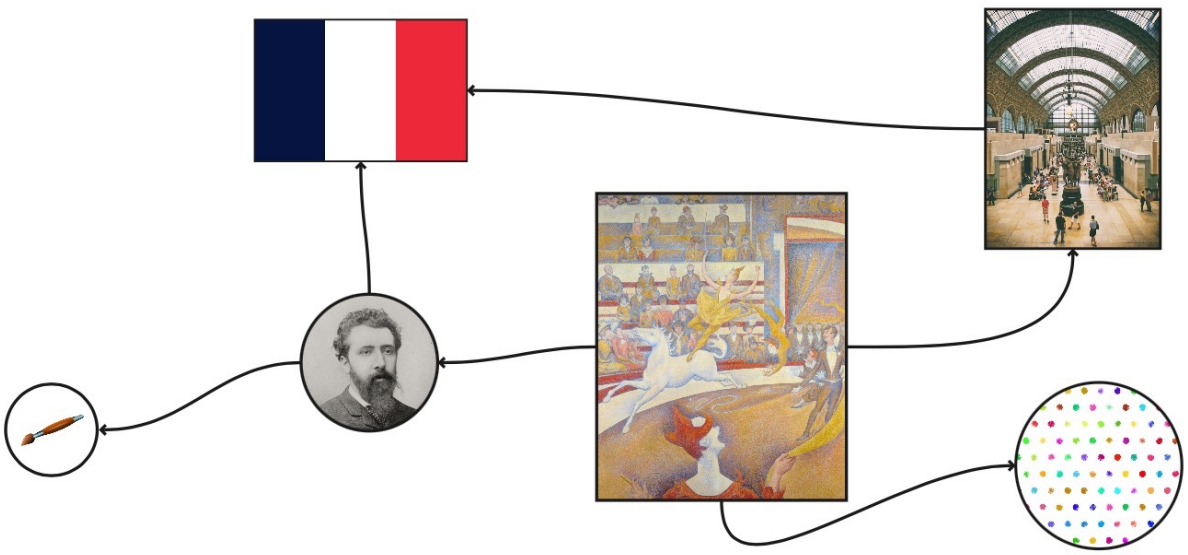

Figure 1.1: Representation of a web of documents without unambiguous indications of what the documents and the links between them represent

Simply put, data coming from different sources can be labeled as Linked Data as soon as they are linked by typed links. In other words, links are no longer just an indication of how to reach another document. Indeed, within the Linked Data story, they also contain information about what exactly the link in question represents. Linked Data thereby ensures the

meaning of data is explicitly defined, in turn rendering the data machine-readable. Figure 1.2 represents the same web of documents as Figure 1.1, but this time in accordance with the idea of Linked Data. Indeed, the documents have been given an unambiguous indication of what they represent, and their mutual semantics have also been clarified thanks to the labeling of their links. (Bizer et al., 2011)

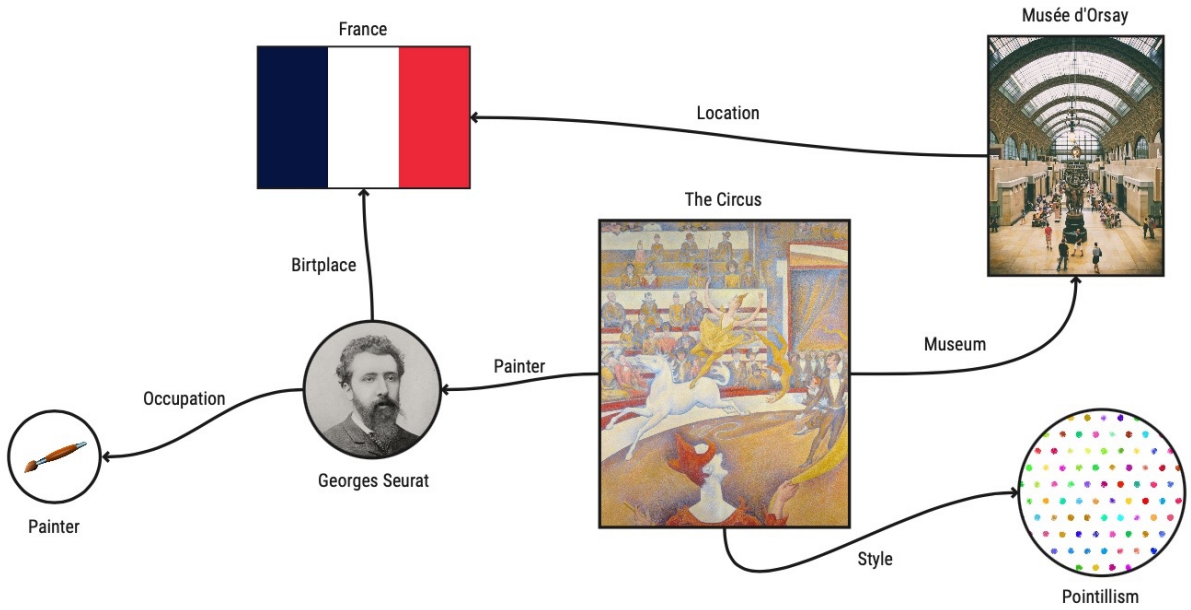

Figure 1.2: Representation of a web of documents composed according to the spirit of Linked Data

Although several technologies exist to achieve the goals of Linked Data, the use of URIs is essential. After all, since URIs are unique, they can unambiguously reference a particular entity. Practically speaking, the URIs that appear in a Linked Data document can be dereferenced using the HTTP protocol in order to retrieve the underlying entities. For instance, `https://stad.gent/id/concept/530010539`, is a URI that can be dereferenced using the HTTP(S) protocol. By dereferencing URI after URI in this way, little by little a - what could be called - *field of information* unfolds, whose semantics can be unambiguously determined by both man and machine. (Bizer et al., 2011)

To clarify the concept of Linked Data, Berners-Lee (2006) put forth four principles to be taken into consideration.

1. **Use URIs as names for things**
   The principle of using URIs has already been discussed above.

2. **Use HTTP URIs so that people can look up those names**
   The principle of using the HTTP protocol to dereference URIs was also touched on above. Nevertheless, it is important to reiterate its importance, as there are other protocols besides HTTP for dereferencing URIs. However, these will technically differ from the HTTP protocol, each in its own different ways. For example, not using the ubiquitous Domain Name System (DNS), is, among others, a common practice among alternative protocols. However, in light of clarity and uniformity, as well as for other technical reasons, the HTTP protocol should be adhered to. (Berners-Lee, 2006)

3. **When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)**
   Obviously, it would not fit within the spirit of Linked Data to obtain a raw data dump when dereferencing a URI that was included from another document as a *Linked Data link*. The obtained data itself must comply with Linked Data principles. Therefore, there are some standards that clearly indicate how ontologies can be described. Consequently, to enable the construction of applications that deal with Linked Data, it goes without saying that a Linked Data document should be built according to the principles of an existing standard. RDF is the most common such standard and is therefore discussed further in Sections 1.1.2. In addition, Section 1.1.4 introduces the SPARQL query interface. After all, large datasets are expected to also provide such interface. (Berners-Lee, 2006)

4. **Include links to other URIs so that they can discover more things**
   The fourth and final principle, too, is rather obvious. After all, by definition, one can only speak of Linked Data when a document refers to at least one other document. In addition, to help advance the cause of transforming the World Wide Web in its current form into a semantic World Wide Web, aided by the concepts of Linked Data, it is preferable to also include links to documents belonging to other sites. (Berners-Lee, 2006)

In conclusion, Linked Data plays a crucial role in giving meaning to the Web by enabling the interconnection and integration of diverse data sources. By adhering to the principles of unique URIs, dereferencing, linking, and using standardized formats, Linked Data fosters a more structured and interconnected web of knowledge. Examples such as DBpedia[1], which provides a structured representation of Wikipedia data, and Friend of a Friend (FOAF), which allows for the description of people and their relationships, illustrate how publishing data as Linked Data benefits from enhanced data discoverability, interlinking with other datasets, and enabling novel applications and insights. Local initiatives like Collections of Ghent (CoGhent[2]), which digitizes art collections from cultural houses in Ghent and will be further discussed in Section 1.4, similarly demonstrate the potential of Linked Data for local organizations in contributing to the broader web of knowledge. (Auer et al., 2007) (Golbeck and Rothstein, 2008) (Van de Vyvere et al., 2022)

1.1.2 Resource Description Framework

The idea behind Linked Data is interesting in itself, but does not yet describe exactly how to get started with it. Therefore, this section introduces the Recourse Description Framework (RDF). Developed under the auspices of the World Wide Web Consortium (W3C), RDF is an infrastructure that allows for the construction of Linked Data datasets and their metadata. Consequently, this not only allows data publishers to lay out their data as Linked Data, but also gives data consumers clear guidance on how the data can be understood. Note here that data consumers can be both individuals and computer applications. (Miller, 1998)

An interesting way to understand RDF is to first make a jump to the English language. Take the sentence below:

The birthplace of Georges Seurat is France.

According to English grammar, the *who* or *what* around which a sentence revolves, is called the subject of the sentence. Therefore, when looking at the sentence above, *Georges Seurat* is its subject. In addition, the part of a sentence that gives

[1] https://www.dbpedia.org
[2] https://www.collections.gent

more information about the subject, is referred to as the predicate, making *the birthplace* the predicate in the above sentence. Finally, the matching value complementing the predicate and completing the sentence, is also of importance. Logically, in the case of the sentence above, that would be *France*. Together, these three components form the most basic building blocks of a sentence. In fact, no matter their lengths, combined, they will always establish a piece of knowledge, exactly what RDF also seeks to accomplish. (Powers, 2003)

The building blocks of RDF data are basically exactly the same as those of linguistic sentences. After all, they are also three in number and even partly share the same names. Moreover, much like with sentences, combined, they form a single yet very clear piece of knowledge. Unlike the English language, however, they are not referred to as sentences. Rather, they are called triples. (Powers, 2003)

- **Resource**
  Miller (1998) defines a resource as any object that is uniquely identifiable by a URI. This enables it to come in different forms: as a web page, as an entire website or simply as any resource on the Web that conveys information in one way or another. (Candan et al., 2001)

  To make the comparison with the English language again, in a triple, the resource corresponds to the subject in a sentence. Moreover, in practice, the term *subject* is often preferred over *resource*. (Powers, 2003)

- **Property Type**
  A property type, or simply a property, introduces a specific aspect, characteristic, attribute, or relationship of a resource. A property type always expects a value to ultimately define the piece of knowledge represented by a triple. (Candan et al., 2001) (Miller, 1998)

  As for property types, in practice, the corresponding term from the English language, *predicate*, is also frequently used as opposed to the more theoretical *property type*. (Powers, 2003)

- **Value**
  A value resolves the concept or relationship initiated by a property type. In this way, it captures the knowledge conveyed by the triple. Values can be represented as text strings, numbers, or any atomic data. However, they can also be resources themselves. This characteristic allows triples therefore to be the building blocks of a web of knowledge. (Miller, 1998)

  It is evident that a value in a triple corresponds to a value in an English sentence. However, in practice, the term *object* is often preferred. (Powers, 2003)

While triples convey a clear and distinct piece of knowledge, a collection of triples can naturally convey a more comprehensive knowledge. Such a collection of triples, interconnected by values that are themselves resources, is also referred to as an *RDF description*. Figure 1.3 illustrates what such an RDF description might look like. Additionally, it is important to note that each of its components, whether it be a resource, property type, or value, does not necessarily have to be a digital concept. After all, Web assets can perfectly represent real-life concepts. (Miller, 1998) (Candan et al., 2001)

Clearly, different terms exist to denote the same RDF concepts. For instance, in addition to the synonyms mentioned above, in literature, the term *statement* is sometimes preferred over *triple*. However, in light of uniformity and clarity, throughout

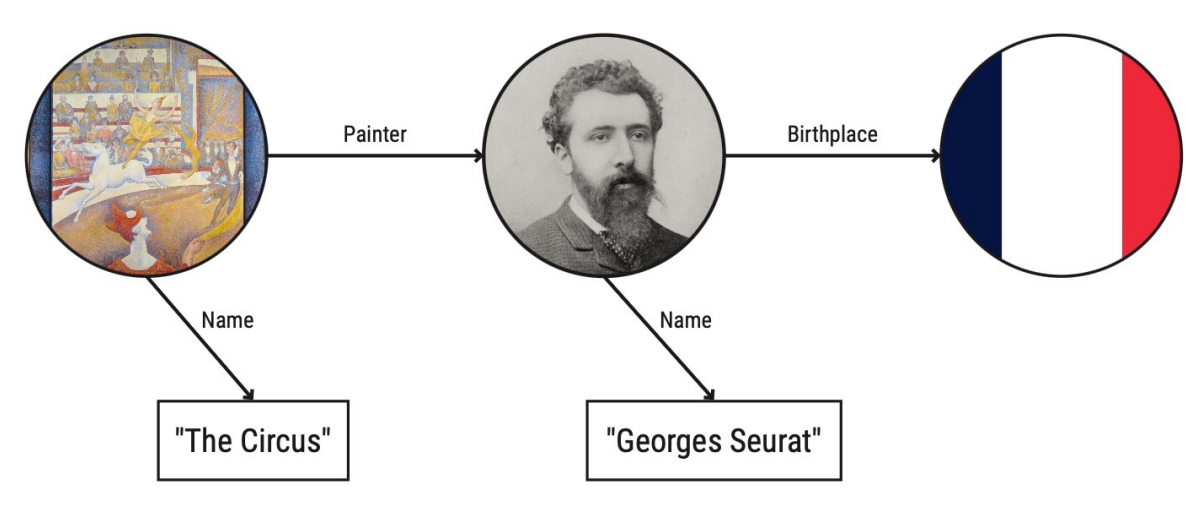

Figure 1.3: Representation of an RDF description

Circles represent resources, arrows represent property types and values are situated at the end of arrows

the rest of this text, the terms *triple*, *subject*, *predicate* and *object* will be used instead of their counterparts. (Candan et al., 2001)

1.1.3 Resource Description Framework Syntax

What constitutes RDF exactly, should be clear by now, but the question of how to actually write down RDF descriptions, still remains to be answered. Therefore, this section introduces some RDF syntaxes. However, since they are not the focus of this research, they will not be discussed in detail. Instead, their outlines will be illustrated by presenting the RDF description from Figure 1.3 in the syntax in question. Incidentally, since the schema presented in Figure 1.3 also has clear guidelines on how to be used, in itself, it also qualifies as an RDF syntax, albeit a graphical one. (Miller, 1998)

All the syntaxes to be discussed are instantiations of the RDF Model and Syntax Specification, providing concrete implementations. However, the first syntax stands apart from the rest as it primarily serves as a notation recommendation for humans to express RDF descriptions in a manner that is unambiguous yet simple. Unlike the other syntaxes, this particular one is not intended for machine consumption. Code Fragment 1.1 demonstrates how the RDF description, as schematically depicted in Figure 1.3, can be represented using this human-centric syntax. In this representation, resources are enclosed in straight brackets, while property types are represented by arrows. Furthermore, the representation of values varies depending on their types. As denoted, resources are encapsulated within brackets. However, if the values are atomic in nature, they are simply enclosed in quotation marks. (Miller, 1998)

The example from Code Fragment 1.1 is easy to read, but at the same time rather confusing. Indeed, certain resource names correspond to certain atomic values. One could of course try to give the resources a more generic name to indicate what exactly the resource in question means. However, that would make little sense given the way the following machine-readable

```
[The Circus] ------name--------> "The Circus"
[The Circus] ------painter-----> [Georges Seurat]
[Georges Seurat] --name--------> "Georges Seurat"
[Georges Seurat] --birthplace--> [France]
```

Code Fragment 1.1: RDF description depicted using a human-centric RDF syntax

RDF syntaxes refer to resources. After all, they use URIs, allowing for a more clear distinction between resources and atomic values.

- **N-Triples**
  Code Fragment 1.2 depicts the representation of the RDF description using N-Triples. In this syntax, each line corresponds to a triple, wherein the subject, predicate, and object are delimited by spaces or tabs. The triple is terminated by a period and a new line character. (Beckett, 2014)

```
<http://example.org/The_Circus> <http://example.org/name> "The Circus" .
<http://example.org/The_Circus> <http://example.org/painter> <http://example.org/Georges_Seurat> .
<http://example.org/Georges_Seurat> <http://example.org/name> "Georges Seurat" .
<http://example.org/Georges_Seurat> <http://example.org/birthplace> <http://dbpedia.org/resource/France> .
```

Code Fragment 1.2: RDF description depicted using the N-Triples syntax

  Furthermore, absolute URIs are employed to denote resources, while atomic values are enclosed within quotation marks. With that in mind, it is important to note that if a value itself contains a quotation mark, it must be properly escaped to ensure correct interpretation. (Beckett, 2014)

- **N3**
  Parsing an RDF description in N-Triples syntax is relatively straightforward for computers, but it can be challenging for humans to comprehend at a glance. The use of absolute URIs in N-Triples can lead to visual clutter and hinder readability. To address this, the N3 syntax builds upon N-Triples by introducing the concept of relative URIs. (Beckett, 2014)

  In N3, it is possible to specify a base URI by including a `@base <URI>` directive at the beginning of the document. When a relative URI is encountered elsewhere in the document, the parser appends it to the specified base URI. This allows for a more concise representation of URIs. (Berners-Lee and Connolly, 2011)

  However, RDF descriptions may contain URIs with different base URIs, making a single base URI insufficient. To overcome this limitation, N3 allows the document to be preceded by one or more `@prefix prefix: <URI>` directives. These directives associate prefixes with URIs, and the parser appends any relative URI preceded by a prefix to the corresponding base URI associated with that prefix. This mechanism enables the use of multiple base URIs within the same document and enhances the flexibility and expressiveness of the N3 syntax. Code Fragment 1.3 illustrates the use of prefixes for the N3 syntax. (Berners-Lee and Connolly, 2011)

- **Turtle**
  The Turtle syntax is very similar to N3. In fact, Turtle is a subset of N3. Specifically, Code Fragment 1.3 can be processed

```
@prefix ex: <http://example.org/> .
@prefix dbp: <http://dbpedia.org/resource/> .

ex:The_Circus ex:name "The Circus" .
ex:The_Circus ex:painter ex:Georges_Seurat .
ex:Georges_Seurat ex:name "Georges Seurat" .
ex:Georges_Seurat ex:birthplace dbp:France .
```

Code Fragment 1.3: RDF description depicted using the N3 and Turtle syntaxes

by a Turtle parser just as well. However, while N3 allows for more expressiveness in principle, Turtle keeps things simpler, making it a popular choice for human readability. (Berners-Lee and Connolly, 2011) (Beckett et al., 2014)

Providing an exhaustive list of the precise differences between the two syntaxes would exceed the scope of this text since the intricacies of RDF syntaxes are not the primary focus here.

- **RDF/XML**

  RDF/XML is one of the earliest RDF syntaxes and remains widely used. To introduce this syntax, Code Fragment 1.4 serves as a guide.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://example.org/"
         xmlns:dbp="http://dbpedia.org/resource/">
  <rdf:Description rdf:about="http://example.org/The_Circus">
    <ex:name>The Circus</ex:name>
    <ex:painter rdf:resource="http://example.org/Georges_Seurat"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/Georges_Seurat">
    <ex:name>Georges Seurat</ex:name>
    <ex:birthplace rdf:resource="http://dbpedia.org/resource/France"/>
  </rdf:Description>
</rdf:RDF>
```

Code Fragment 1.4: RDF description depicted using the RDF/XML syntax

The RDF description in RDF/XML is enclosed within `rdf:RDF` elements, where necessary prefixes can also be defined. While an XML declaration like `<?xml version="1.0"?>` can precede the RDF/XML document, it is optional and omitted in Code Fragment 1.4 to focus primarily on the basics of RDF syntaxes. (Gandon et al., 2014)

Upon encountering the `rdf:RDF` tag, a parser recognizes that it should process an RDF description. In RDF/XML, such an RDF description is constructed using one or more `rdf:Description` elements. In fact, each `rdf:Description` element represents a subject, and its optional `rdf:about` attribute denotes the subject's URI. Consequently, the triples associated with the subject are enclosed within the corresponding `rdf:Description` tags. Predicates on the one hand, whether represented using a prefix or not, have their own elements. The representation of subjects, on the other hand, depends on their nature: for atomic values, they can simply be placed between opening and closing

subject tags, while for resource subjects, their URIs are included as the value of an `rdf:resource` attribute within the subject tag. (Gandon et al., 2014)

Once again, it is important to note that the Code Fragments used in this section provide only an introductory glimpse of the proposed syntaxes. They cover only a small portion of the potential scope of a syntax. Code Fragment 1.4, in particular, demonstrates that RDF/XML syntax can obscure simplicity, especially when dealing with more extensive RDF descriptions. Consequently, RDF/XML is not commonly used for human-readable purposes but rather as a syntax primarily intended for machine consumption. (Dongo and Chbeir, 2019)

- **JSON-LD**

  The final RDF syntax introduced is called JSON-LD. Similar to RDF/XML, JSON-LD builds upon an existing syntax for representing data on the web. However, JSON-LD representations are generally more human-readable. As most resources and examples in the following text will be presented in JSON-LD, a slightly more comprehensive overview of this syntax is provided compared to the previous ones. Nevertheless, what follows is not an exhaustive listing of all the intricacies of the syntax. Instead, it aims to offer readers a concise introduction to JSON-LD without prior knowledge, making the rest of the text more easily comprehensible. For those seeking more in-depth information about JSON-LD, it is recommended to consult other sources[3].

  It is evident that the same data can be represented in various ways, and this applies to RDF data as well. While the visual representation of an RDF description, as depicted in Figure 1.3, is relatively straightforward, converting it into a fully textual format poses certain choices to be made. After all, there are numerous possibilities regarding the exact data representation. In the introduction of previous syntaxes, a specific representation was chosen each time. However, in this section, three different approaches for representing the same set of data using the JSON-LD syntax are presented.

  To start off, Code Fragment 1.5 closely resembles the previous examples, using nesting to store all the data in a single JSON-LD document. However, some may question whether it is appropriate to make the `George_Seurat` resource a child of `The_Circus` resource, implying a hierarchical relationship that may not be relevant.

  Subsequently, in Code Fragment 1.6, the data is split into two JSON-LD documents. Utilizing URIs, the documents can still refer to each other uniquely, without suggesting any hierarchical relationship between the resources.

  Finally, Code Fragment 1.7 takes a distinct approach by using the `@graph` property. This allows listing the necessary resources in a JSON array, placing them on equal footing within a single document. However, this method introduces extra clutter and overhead compared to the previous approaches. (Sporny et al., 2020)

  Ultimately, the choice of representation depends on the specific use case and the desired balance between simplicity and expressiveness. Each approach has its advantages and trade-offs, showcasing the flexibility of the JSON-LD syntax in accommodating different data representation needs.

  Understanding Code Fragments 1.5, 1.6, and 1.7 becomes relatively straightforward after having discussed the previous syntaxes. However, two aspects deserve further attention: the use of `@id` and `@context` keywords in JSON-LD.

[3] The W3C JSON-LD 1.1 Recommendation provides very in-depth information about the JSON-LD syntax: https://www.w3.org/TR/json-ld11/.

```
{
  "@context": {
    "ex": "http://example.org/",
    "dbp": "http://dbpedia.org/resource/"
  },
  "@id": "ex:The_Circus",
  "ex:name": "The Circus",
  "ex:painter": {
    "@id": "ex:Georges_Seurat",
    "ex:name": "Georges Seurat",
    "ex:birthplace": "dbp:France"
  }
}
```

Code Fragment 1.5: RDF description with nested objects depicted using the JSON-LD syntax

```
Document 1:
{
  "@context": {
    "ex": "http://example.org/"
  },
  "@id": "ex:The_Circus",
  "ex:name": "The Circus",
  "ex:painter": "ex:Georges_Seurat"
}

Document 2:
{
  "@context": {
    "ex": "http://example.org/",
    "dbp": "http://dbpedia.org/resource/"
  },
  "@id": "ex:Georges_Seurat",
  "ex:name": "Georges Seurat",
  "ex:birthplace": "dbp:France"
}
```

Code Fragment 1.6: RDF description spread over two documents depicted using the JSON-LD syntax

```
{
  "@context": {
    "ex": "http://example.org/",
    "dbp": "http://dbpedia.org/resource/"
  },
  "@graph": [
    {
      "@id": "ex:The_Circus",
      "ex:name": "The Circus",
      "ex:painter": {
        "@id": "ex:Georges_Seurat"
      }
    },
    {
      "@id": "ex:Georges_Seurat",
      "ex:name": "Georges Seurat",
      "ex:birthplace": {
        "@id": "dbp:France"
      }
    }
  ]
}
```

Code Fragment 1.7: RDF description as a graph depicted using the JSON-LD syntax

Firstly, the `@id` keywords uniquely identify the proposed resources using URIs. Indeed, in the given examples, the id's do exactly that. (Sporny et al., 2020)

Secondly, the `@context` keyword plays a crucial role in JSON-LD. It introduces specifics that can be taken for granted in the actual data, reducing the need for repetitive information and cleaning up the actual JSON. While Code Fragments 1.5, 1.6, and 1.7 use the context in a straightforward way by introducing prefixes, in practice, it can do more than that. Essentially, the context maps terms to URIs. These terms can be freely chosen to enhance human readability. (Sporny et al., 2020)

W3C's JSON-LD Recommendation[4] offers a valuable example of how the context is typically used, as illustrated in Code Fragment 1.8. The provided context clearly indicates that when the key `name` appears in the data, it refers to `http://schema.org/name`. Similarly, for `image` and `homepage`, their respective values are *expanded* into objects that hold additional information. The `@type` keyword is also used in the example to indicate the type of the final value. In Code Fragment 1.8, it shows that the `image` and `homepage` keys are followed by an `@id`, representing unique resources. Moreover, JSON-LD supports various other types, and custom types can be defined to suit specific requirements. (Sporny et al., 2020)

```
{
  "@context": {
    "name": "http://schema.org/name",
    "image": {
      "@id": "http://schema.org/image",
      "@type": "@id"
    },
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  },
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

Code Fragment 1.8: Example of context use in JSON-LD, proposed by Sporny et al. (2020)

To further enhance the cleanliness of a JSON-LD document, one can opt to store the context as a separate resource rather than embedding it directly in the document. Using this approach, the JSON-LD document includes the URI that references the context as the value for the `@context` key. Storing the context separately allows for greater modularity and reusability, making it easier to manage and maintain complex JSON-LD documents. The use of separate contexts can significantly improve the organization and readability of JSON-LD data, enhancing its compatibility with RDF and Linked Data principles. (Sporny et al., 2020)

[4] https://www.w3.org/TR/json-ld11/

To finish off this section on JSON-LD, it is interesting to note that when the JSON-LD document presented in Code Fragment 1.8 is *expanded*, the data takes on its typical RDF form, adhering fully to the Linked Data principles. This expansion, as shown in Code Fragment 1.9, reveals the underlying structure of the data and its connection to other resources. (Sporny et al., 2020)

```
[{
  "http://schema.org/name": [{"@value": "Manu Sporny"}],
  "http://schema.org/url": [{ "@id": "http://manu.sporny.org/" }],
  "http://schema.org/image": [{ "@id": "http://manu.sporny.org/images/manu.png" }]
}]
```

Code Fragment 1.9: Example of an expanded JSON-LD document, proposed by Sporny et al. (2020)

In summary, the `@id` and `@context` keywords in JSON-LD contribute to the readability, expressiveness, and flexibility of representing RDF data, enabling a more human-friendly approach to data serialization.

Before concluding this section on RDF syntaxes, it is crucial to reiterate that the explanations provided are not exhaustive. Only a surface-level overview of these syntaxes was covered, and there is much more to explore and learn about them. This section serves as a reference for those with limited or no prior knowledge of RDF syntaxes, aiming to facilitate their understanding of the remaining text. In the following sections, several RDF examples will be presented, with the majority of them using the Turtle and JSON-LD syntaxes. However, there will be no further elaboration on new elements that are specific to each syntax unless they are essential for a clear understanding of the text. For readers seeking a more in-depth understanding of the syntaxes, additional resources are recommended to further explore their intricacies and capabilities.

1.1.4 SPARQL

SPARQL is a set of specifications that describes how to work with RDF data. The latest version of SPARQL is SPARQL `1.1`, which, among others, stipulates the workings of an update language, query results formats, and federated querying. But arguably most importantly, it defines the SPARQL query language. (Buil-Aranda et al., 2013)

The SPARQL query language is designed for querying RDF data sources. While `CONSTRUCT` queries return results as new RDF data, queries with a `SELECT` clause return specific data points. This research exclusively focuses on the latter type of queries. (Seaborne and Harris, 2013)

The `SELECT` clause specifies which variables - in their original form or modified - should be returned as results from the `WHERE` clause. The `WHERE` clause in turn defines the *basic graph pattern* (BGP) that the datasource(s) need to match. Such a BGP consists of one or more triple patterns that are matched one by one with the triples from the queried dataset(s). Triple patterns are similar to regular triples, but the subject, predicate, and/or object can be replaced with a variable. When a triple pattern matches a triple, each of its variables is combined with the value of the corresponding triple's corresponding element, into a *binding*. In subsequent triple patterns, previously encountered variables can reappear, allowing their corresponding bindings to already narrow down the list of possible matching triples. (Seaborne and Harris, 2013)

The `SELECT` and `WHERE` clauses are essential to SPARQL queries, but SPARQL provides many other keywords to further specify queries. For instance, `FILTER` statements can subject variable values to additional tests, wrapping certain triple

patterns in an `OPTIONAL` clause alleviates them from necessarily being matched, and requesting only unique results can be done using `DISTINCT`. More advanced options include merging different result sets using the `UNION` keyword, aggregating results with a `GROUP BY` statement, and manipulating variable values on the spot using a `BIND` form. Basic query necessities like setting a limit (`LIMIT`) and an offset (`OFFSET`) are also available. Finally, queries can be made more readable and organized by using `PREFIX` statements at the top of the query, preventing the need of writing out full URIs. In conclusion, the use and combination of any of these keywords make it possible to craft a wide range of queries. The queries presented throughout this research can generally be considered *simple* and should therefore be comprehensible to readers who are new to the subject. However, for those interested, Section 1.4.2 already provides some example queries to explore. (Seaborne and Harris, 2013) (DuCharme, 2013)

Up to this point, the terms *triple* and *triple pattern* have been used exclusively. However, it is important to note that in literature, the terms *quad* and *quad pattern* also often appear. Essentially, quads are the same as triples but they introduce a fourth element, namely a *named graph*. In fact, these named graphs *group* certain triples and allow datasets to be subdivided further. This research does not further discuss nor employ named graphs, yet since the term *quad* is more specific, from this point onward, it will be used in favor of the term *triple*. (Taelman, 2020)

1.2 Link-Traversal-based Query Processing

TODO

1.2.1 Link Traversal Basics

TODO

1.2.2 Reachability Criteria

TODO

1.3 Comunica

TODO

1.3.1 Building Blocks

TODO

1.3.2 Link Traversal Engines

TODO

1.4 Collections of Ghent

TODO

1.4.1 Linked Data Event Streams

TODO

1.4.2 Example Queries

TODO

1.4.3 Query Builder

TODO (see Figure 1.4)

1.5 International Image Interoperability Framework

TODO

1.5.1 IIIF Manifests

TODO

1.5.2 IIIF Viewers

TODO (see Figure 1.5)

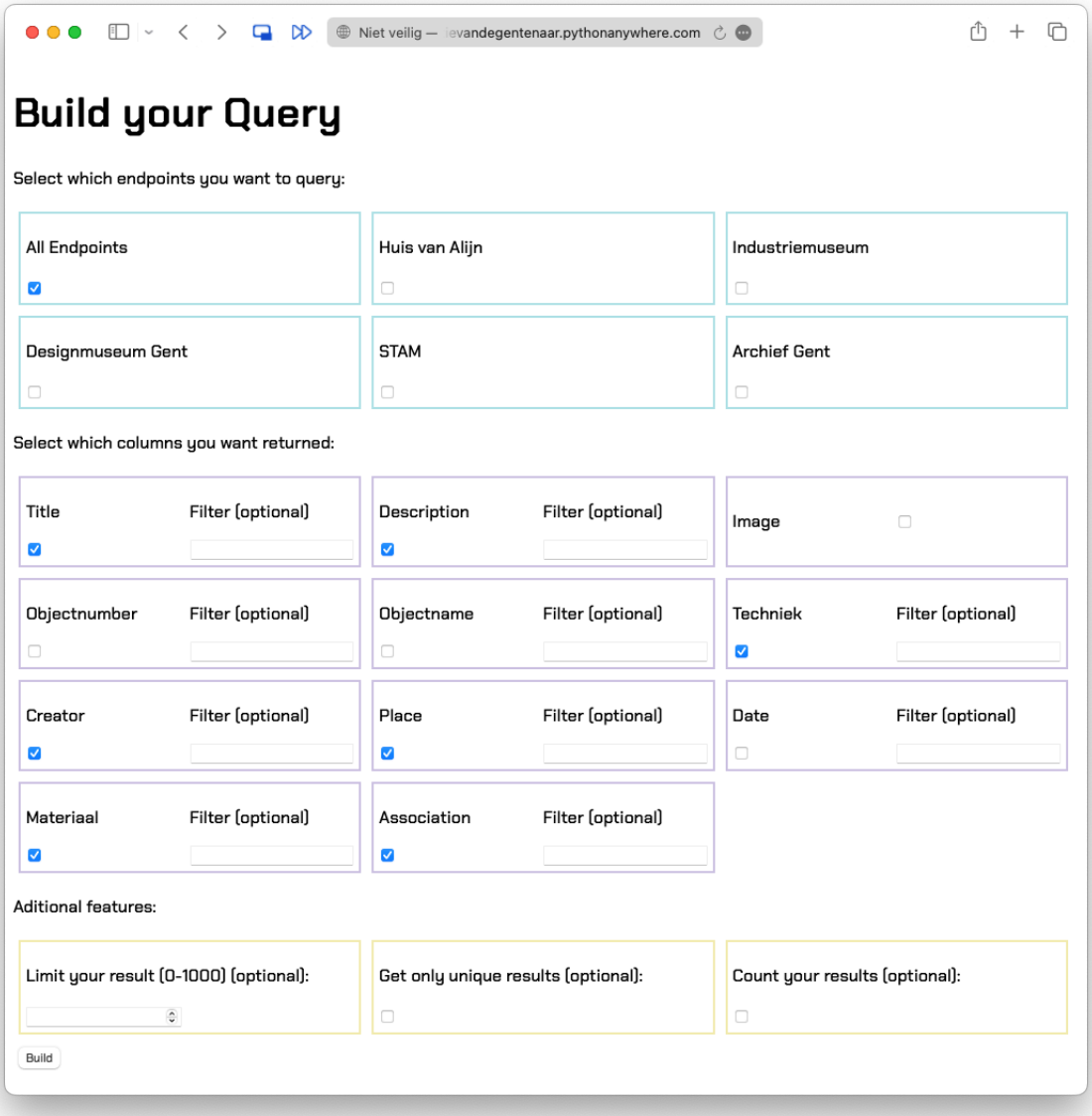

Figure 1.4: Screenshot of CoGhent Query Builder

```
PREFIX cidoc:<http://www.cidoc-crm.org/cidoc-crm/>
PREFIX adms:<http://www.w3.org/ns/adms#>
PREFIX skos:<http://www.w3.org/2004/02/skos/core#>
PREFIX la:<https://linked.art/ns/terms/>

SELECT ?title ?note ?associatie ?creator ?plaats ?techniek ?materiaal

WHERE {
    # Title
    ?o cidoc:P102_has_title ?title.

    # Description
    ?o cidoc:P3_has_note ?note.

    # Association
    ?o cidoc:P128_carries ?carries.
    ?carries cidoc:P129_is_about ?about.
    ?about cidoc:P2_has_type ?type.
    ?type skos:prefLabel ?associatie.

    # Creator
    ?o cidoc:P108i_was_produced_by ?production.
    ?production cidoc:P14_carried_out_by ?producer.
    ?producer la:equivalent ?equivalent.
    ?equivalent rdfs:label ?creator.

    # Place
    ?o cidoc:P108i_was_produced_by ?produced.
    ?produced cidoc:P7_took_place_at ?tookplace.
    ?tookplace la:equivalent ?plaatsequivalent.
    ?plaatsequivalent skos:prefLabel ?plaats.

    # Technique
    ?o cidoc:P108i_was_produced_by ?produced.
    ?produced cidoc:P32_used_general_technique ?technique.
    ?technique cidoc:P2_has_type ?hastype.
    ?hastype skos:prefLabel ?techniek.

    # Material
    ?o cidoc:P45_consists_of ?consists.
    ?consists cidoc:P2_has_type ?materiaaltype.
    ?materiaaltype skos:prefLabel ?materiaal.
}
```

Code Fragment 1.10: Example of SPARQL query created by original CoGhent Query Builder

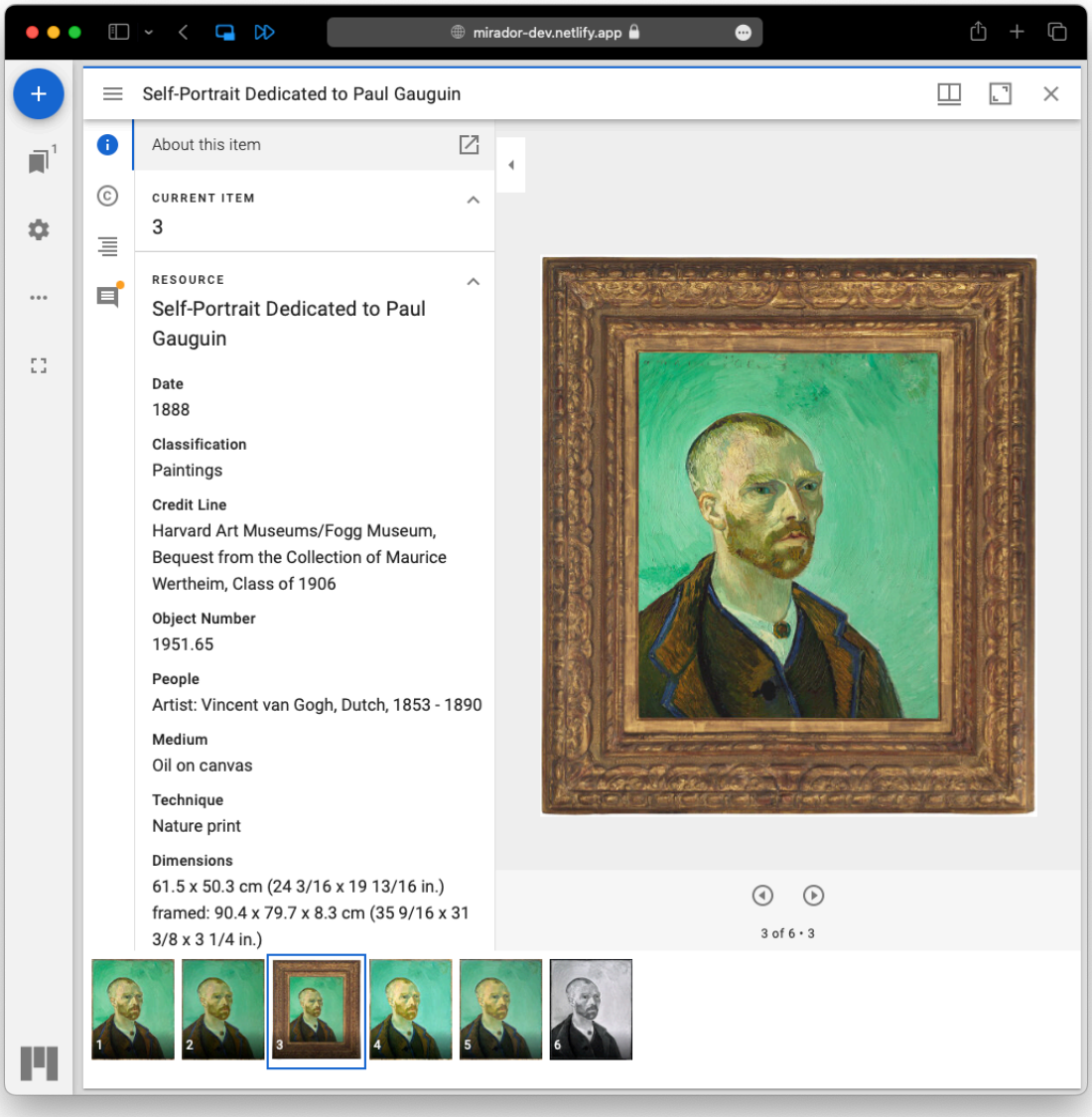

Figure 1.5: Screenshot of Mirador IIIF Viewer

2

CoGhent Data and Link Traversal

The primary focus of this research is the development of tools for constructing queries that target specific properties of CoGhent Human-Made Objects. These queries can either be confined to data within the CoGhent LDESs or extend beyond them by employing Link Traversal to follow links and traverse the corresponding documents. This approach facilitates the acquisition of new insights into the CoGhent data by not only enhancing the understanding of specific Human-Made Objects but also enabling their comparison in novel ways.

In the subsequent sections of this research, Comunica's link traversal capabilities will be utilized, as its modularity allows for the creation of link traversal engines tailored to the structure of the CoGhent data and the specific needs of this research. However, it is important to note that link traversal, despite its potential, remains an active area of research and can be configured in various ways.

This chapter therefore aims to explore the use of link traversal for discovering properties of Human-Made Objects, starting from the CoGhent LDESs. The chapter begins by providing an overview of the available data sources that can serve as starting points for the link traversal process. It then delves into the development of a link traversal engine optimized for the objectives outlined above. Finally, the chapter examines the most pertinent and intriguing types of resources to which the CoGhent Human-Made Objects link. These resources will be crucial for achieving the goal of broadening the knowledge of the CoGhent data.

2.1 CoGhent Data Sources

CoGhent provides a set of LDESs for each participating institution. These LDESs are accessible through specific endpoints, as listed in Table 2.1

2.1.1 URI Redirection

When accessing any of the URIs listed in Table 2.1, it is resolved to the same URI but with an additional query parameter `generatedAtTime`. For example, accessing the LDES from Industriemuseum results in the original URI being extended with `?generatedAtTime=2023-08-17T00:07:32.016Z`[1].

[1] Since the query parameter's value is time-dependent, this specific value serves only as an example of how it is structured.

Table 2.1: CoGhent LDES endpoints as published by CoGhent (2022)

Publishing organisation	Endpoint URI
Design Museum Gent (DMG)	https://apidg.gent.be/opendata/adlib2eventstream/v1/dmg/objecten
Huis van Alijn (HVA)	https://apidg.gent.be/opendata/adlib2eventstream/v1/hva/objecten
Industriemuseum	https://apidg.gent.be/opendata/adlib2eventstream/v1/industriemuseum/objecten
STAM	https://apidg.gent.be/opendata/adlib2eventstream/v1/stam/objecten
Archief Gent	https://apidg.gent.be/opendata/adlib2eventstream/v1/archiefgent/objecten

This behavior is confirmed by running the following command:

```
curl -i "https://apidg.gent.be/opendata/adlib2eventstream/v1/industriemuseum/objecten"
```

This returns an HTTP `302 Found` response code and a `Location` header with the extended URI, indicating a redirect to that link. Eventually, when a client (e.g. a browser or Comunica) sends a `GET` request to the updated link, the server returns the last (most recent) page of the requested LDES in JSON-LD format. (MDN Web Docs, 2023)

2.1.2 Non-deterministic results

When configuring a query engine, any or multiple of the CoGhent endpoints can be chosen as data sources, depending on the specific data of interest. Naturally, due to the nature of LDESs, the same query should never be assumed to yield the same results across multiple executions. It is essential to understand that, in theory, link traversal engines should produce deterministic results in both content and order. In practice, however, this deterministic nature is often disrupted. The timing of HTTP responses, crucial for fetching documents, can introduce variability. Even if the LDESs remain unchanged, these responses can arrive at varied intervals, affecting the engine's predetermined processing order.

This phenomenon is demonstrated by running the query displayed in Code Fragment 2.1[2] twice, using Design Museum Gent's LDES as data source and making sure it does not get updated during the experiment. Tables 2.2 and 2.3 show, for both executions respectively, each result's IIIF Manifest URI, as well as the order in which the results were returned. Comparing both outputs clearly proves the results from the two executions differ in both content and order.

For similar reasons, the order in which CoGhent endpoint URIs are given to the engine as data sources, in practice does not necessarily imply that one endpoint's data has priority over the other. This is illustrated by running the same query (see Code Fragment 2.1) with the Design Museum Gent LDES first and the Huis Van Alijn LDES second, and then reversing the order. The results from both executions, as shown in Tables 2.4 and 2.5 respectively, once again show variations in content and order, yet most importantly do not seem to show any notable correlation to the order in which the endpoints were given to the engine.

[2] The query's specifics are discussed in Section 2.3.1.

```
PREFIX iiif: <http://iiif.io/api/presentation/2#>
PREFIX cidoc:<http://www.cidoc-crm.org/cidoc-crm/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX w3-exif: <http://www.w3.org/2003/12/exif/ns#>
PREFIX w3-oa: <http://www.w3.org/ns/oa#>

SELECT ?manifest ?height ?image

WHERE {
  # Manifest URI
  ?human_made_object cidoc:P129i_is_subject_of ?manifest.

  # Image height
  ?manifest iiif:hasSequences/rdf:first/iiif:hasCanvases/rdf:first/w3-exif:height ?height.

  # Image URI
  ?canvas iiif:hasImageAnnotations/rdf:first/w3-oa:hasBody ?image.
}

LIMIT 10
```

Code Fragment 2.1: SPARQL query fetching ten Human-Made Object's IIIF Manifest URIs, image heights and image file URIs

Table 2.2: (Part of) results after **first** execution of query displayed in Code Fragment 2.1

	IIIF Manifest URI
1	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:3086_3-5
2	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:1992-0068
3	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:3130
4	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:1990-0051_0-5
5	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:3054
6	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:3124
7	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0284
8	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0296
9	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0305
10	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0281_21-21

Table 2.3: (Part of) results after **second** execution of query displayed in Code Fragment 2.1

	IIIF Manifest URI
1	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:3075
2	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0305
3	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:3054
4	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:1563
5	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:1987-0447
6	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:1987-1127_1-2
7	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0271
8	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0284
9	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2018-0296
10	https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:2990_0-4

Table 2.4: (Part of) results after execution of query displayed in Code Fragment 2.1 with Design Museum Gent (**DMG**) LDES endpoint as **first** data source and Huis Van Alijn (**HVA**) LDES endpoint as **second** data source

	IIIF Manifest URI
1	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2014-031-015
2	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2015-024-001
3	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:3223
4	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:3086_3-5
5	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:1563
6	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2014-031-001
7	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:1987-1127_2-2
8	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2014-031-002
9	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:1987-0447
10	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2014-031-003

Table 2.5: (Part of) results after execution of query displayed in Code Fragment 2.1 with Huis Van Alijn (**HVA**) LDES endpoint as **first** data source and Design Museum Gent (**DMG**) LDES endpoint as **second** datasource

	IIIF Manifest URI
1	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2014-031-002
2	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2014-031-001
3	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2014-031-003
4	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2009-018-568
5	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2009-018-568
6	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2015-024-004
7	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:2018-0261
8	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:2990_4-4
9	https://api.collectie.gent/iiif/presentation/v2/manifest/**dmg**:2018-0260
10	https://api.collectie.gent/iiif/presentation/v2/manifest/**hva**:2015-024-001

2.1.3 Duplicate Human-Made Objects

It is also important to note that, since updates to an LDES object are performed by adding a new version of the object to the LDES, it is possible to receive multiple results for the same Human-Made Object. As discussed in Secion 1.4.2, a potential workaround would be to use a combination of `distinct` and `order by` clauses in the query itself, to only retrieve the newest versions. However, since ordering can only occur when all results are in, this approach prevents them from appearing in a *streaming* manner. A more efficient solution, therefore, is to let the application that initiated the query, keep track of Human-Made Object URIs while the results are coming in. That way, when the application encounters duplicate Human-Made Objects, it can decide to only retain the latest version's results. Since implementing such a solution is considered trivial, the issue will not be discussed further in this research.

2.1.4 Conclusion

In conclusion, the CoGhent LDES endpoints work perfectly well to initiate the Link Traversal-based Querying process from. Each institution having a separate LDES is an added bonus, as this gives users the flexibility to choose which institutions' data to query. However, it is essential to be aware that the results and order of results are not predictable due to the nature of LDESs as well as Comunica's LTQP implementation. Additionally, Human-Made Objects are spread over multiple pages in the LDES, which needs to be taken into consideration when building the Comunica link traversal engine configuration.

2.2 Comunica Link Traversal Engine Configuration

As discussed in Section 1.3, the Comunica engine offers a wide range of configurability for link traversal. Numerous link traversal-specific actors have been developed. Some of those have already matured, while others are still in active development. In this section, some of these actors will be considered for configuring a Comunica link traversal engine that meets

the requirements of this research, as well as performs up to a standard that is acceptable for real-world use. The resulting configuration should ultimately determine the engine used throughout the rest of this research.

2.2.1 Base Configuration

The Comunica Link Traversal repository[3] already provides several predefined configurations[4] that are *out of the box* available to Comunica users to kick-start with LTQP. A common feature of these configurations is the initial import of `config-base.json`[5]. This configuration file imports all actors and mediators necessary for the basic functionality of a Comunica Link Traversal engine, such as HTTP fetching, query operations, and RDF parsing. In other words, such a base configuration is essential to having a working link traversal engine. However, since this research does not focus on these basic functionalities, the intricacies of setting up a base configuration will not be discussed further. Rather, as is the case with the predefined configurations, the configuration specific to this research will also start with importing the `config-base.json` file.

2.2.2 Basic Link Extractors

The most important type of actors that should be considered when setting op a link traversal engine, are arguably the link extractors. When a new RDF document is encountered during the link traversal process, these actors determine which links from that document should be added to the link queue. In other words, they are the ones deciding which resources should be queried.

The most basic link extractor is the *All Extract Links Actor*[6]. This actor essentially implements the *cAll* criterion, as discussed in Section 1.2. Simply put, it adds all links it encounters to the link queue. However, this approach is not suitable for the purposes of this research, as it may lead to traversing too many documents that will most certainly not aid in resolving the query at hand, in turn leading to impractical execution times.

As already discussed, this research focuses on queries that fetch data specific to Human-Made Objects. This means that the specific paths to follow - starting from a Human-Made Object and ending in the object of interest - are known beforehand. In other words, the queries already specify these *sequences of predicates*, allowing for a more targeted approach. Therefore, another interesting link extractor to consider, is the *Quad Pattern Query Extract Links Actor*[7]. Essentially, this actor is an implementation of the *cMatch* criterion that was discussed in Section 1.2. It adds only those links to the link queue that are part of quads that match at least one quad pattern in the query. Given the knowledge of the starting subject - Human-Made objects - and the specific sequence of predicates to follow, this actor should better *guide* the engine in the right direction, leading to faster results. However, it is possible for certain documents to, by change, contain quads that do not lead to the data the query was set up for, still leading to *wrong* documents being visited.

[3] https://github.com/comunica/comunica-feature-link-traversal
[4] https://github.com/comunica/comunica-feature-link-traversal/tree/master/engines/config-query-sparql-link-traversal/config
[5] https://github.com/comunica/comunica-feature-link-traversal/blob/master/engines/config-query-sparql-link-traversal/config/config-base.json
[6] https://github.com/comunica/comunica-feature-link-traversal/tree/master/packages/actor-extract-links-all
[7] https://github.com/comunica/comunica-feature-link-traversal/tree/master/packages/actor-extract-links-quad-pattern-query

2.2.3 Extracting Links based on Predicates

Having in mind that sequences of predicates are already known beforehand, the most promising link extractor is the *Predicates Extract Links Actor*[8]. This type of link extractor was not discussed before, but its workings are straightforward. Essentially, for every quad in a document, the actor only considers objects. Apart from the object naturally needing to be a URI, the only links that are added to the link queue are those objects' links that have a predicate matching one of the regexes set in the actor's configuration. In other words, the sequences of predicates that define the queries considered in this research, can literally serve as the regexes this actor uses to evaluate predicates. Additionally, the *rules* can even be tightened by obliging every quad's subject to match the URI of the document currently being processed. This extra requirement further narrows down the selection of links to follow, potentially speeding up the querying process even further.

To test this approach, a Comunica link traversal query engine is built using the configuration as depicted in Code Fragment 2.2, in turn tasked with resolving the query displayed in Code Fragment 2.1. Once again, the data source is set to the Design Museum Gent LDES endpoint. As can be seen in Code Fragment 2.2, the configuration's second import is a custom configuration file. This file is displayed in Code Fragment 2.3 and not only tasks the engine being built to use the Predicates Extract Links Actor, but also instructs this link actor to only consider object links whose predicates match the query's predicates and whose subjects match the current document's URI. The keys that specify these settings are respectively called `predicateRegexes` and `checkSubject`.

```
{
  "@context": [
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        config-query-sparql/^2.0.0/components/context.jsonld",
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        config-query-sparql-link-traversal/^0.0.0/components/context.jsonld"
  ],
  "import": [
    "ccqslt:config/config-base.json",
    "./actors/extract-links-predicates-custom.json"
  ]
}
```

Code Fragment 2.2: Custom link traversal engine configuration using Predicates Extract Links Actor

However, after building the engine and instructing it to resolve the query, no results are returned. To uncover the reason for this failure, the logs[9] outputted by the engine during execution and displayed in Code Fragment 2.4, can be consulted. From these logs, it can be inferred that the engine initially fetches the provided data source, in this case, the Design Museum Gent LDES. Then, it retrieves the documents referenced in the context of the LDES in order to expand the LDES. Finally, once this expansion is completed successfully, the LDES is marked as *identified*. As for the rest of the logs, there are no significant actions taking place. In other words, no other documents are identified, let alone requested. From this, it can be deduced that

[8] https://github.com/comunica/comunica-feature-link-traversal/tree/master/packages/actor-extract-links-predicates
[9] Logging can be enabled as explained here: https://comunica.dev/docs/query/advanced/logging/.

```
{
  "@context": [
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        runner/^2.0.0/components/context.jsonld",
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        actor-extract-links-predicates/^0.0.0/components/context.jsonld"
  ],
  "@id": "urn:comunica:default:Runner",
  "@type": "Runner",
  "actors": [
    {
      "@id": "urn:comunica:default:extract-links/actors#predicates-common",
      "@type": "ActorExtractLinksPredicates",
      "checkSubject": true,
      "predicateRegexes": [
        "http://www.cidoc-crm.org/cidoc-crm/P129i_is_subject_of",
        "http://iiif.io/api/presentation/2#hasSequences",
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#first",
        "http://iiif.io/api/presentation/2#hasCanvases",
        "http://www.w3.org/2003/12/exif/ns#height",
        "http://iiif.io/api/presentation/2#hasImageAnnotations",
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#first",
        "http://www.w3.org/ns/oa#hasBody"
      ]
    }
  ]
}
```

Code Fragment 2.3: Comunica Predicates Extract Links Actor configuration with predicate regexes set to predicates from query displayed in Code Fragment 2.1 and subject checking **enabled**

no links are being added to the link queue while traversing the LDES. This suggests that the configuration of the Predicates Extract Links Actor needs to be reviewed.

Through debugging, it can be determined that only two quads pass the test comparing their subject URIs to the URI of the current document, in this case the LDES. These quads in question are both TREE-related quads - as mentioned in Section 1.4.1, LDESs are built on the TREE specification. It comes as no surprise that these quads fail the subsequent test that compares predicates with the provided regexes. However, the fact that only these two quads pass the first test, and every other quad fails, highlights why the configuration of the Predicates Extract Links Actor, as shown in Code Fragment 2.3, does not work for the query presented in Code Fragment 2.1: since the *starting point* of the query is expected to be a Human-Made Object subject - the first predicate `cidoc:P129i_is_subject_of` achieves this as only Human-Made Object subjects have this predicate in the LDES - these subjects will never match the URI of the LDES. As a result, the Predicates Extract Links Actor will disregard these quads.

One possible solution is to modify the query by providing the LDES page itself as the *starting point* and extending the sequence of predicates to *bridge the gap* between the LDES root node and the Human-Made Objects. However, as part of the aim of this research is to assist people without a technical background in constructing and better comprehending queries, making the queries unnecessarily long and complex is not desirable. Consequently, the decision is made to set the `checkSubject` key in the configuration of the Predicates Extract Links Actor to `false`. This ultimately leads to the configuration presented in Code Fragment 2.5.

2.2.4 Comparing Link Extractors

In an attempt to compare the discussed link extractors not only in terms of functionality but also in terms of performance, a small experiment is conducted. Similar to before, the query shown in Code Fragment 2.1 is used, with the Design Museum Gent LDES serving as the data source. The first engine utilizes the All Extract Links Actor, the second one employs the Predicates Extract Links Actor, and the third utilizes the Predicates Extract Links Actor in the configuration outlined in Code Fragment 2.5. Consequently, the final engine configurations corresponded to the existing *Follow All*[10] and *Follow Match Query*[11] configurations present in the Comunica Link Traversal GitHub repository, along with the custom configuration as illustrated in Code Fragment 2.2. To ensure reliability, each engine executes the query consecutively three times, with the engine's complete HTTP cache being invalidated after each run. The outcomes of the experiment are presented in Table 2.6.

Table 2.6: Results from experiment comparing different Comunica link traversal engines

Engine	Total time (s)	Average time single execution (s)
Follow All	Runtime error	Runtime error
Follow Match Query	66.08	22.03
Custom (using configuration displayed in Code Fragment 2.5)	54.34	18.11

[10] https://github.com/comunica/comunica-feature-link-traversal/blob/master/engines/config-query-sparql-link-traversal/config/config-follow-all.json

[11] https://github.com/comunica/comunica-feature-link-traversal/blob/master/engines/config-query-sparql-link-traversal/config/config-follow-match-query.json

```
[...]  INFO: Requesting
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             dmg/objecten
             { ... , method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }

...

[...]  INFO: Requesting
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             context/cultureel-erfgoed-object-ap.jsonld
             { ..., method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }
[...]  INFO: Requesting
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             context/persoon-basis.jsonld
             { ..., method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }
[...]  INFO: Requesting
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             context/cultureel-erfgoed-event-ap.jsonld
             { ..., method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }
[...]  INFO: Requesting
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             context/organisatie-basis.jsonld
             { ..., method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }
[...]  INFO: Requesting
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             context/generiek-basis.jsonld
             { ..., method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }
[...]  INFO: Requesting
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             context/dossier.jsonld
             { ..., method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }
[...]  INFO: Identified as file source:
             https://apidg.gent.be/opendata/adlib2eventstream/v1/
             dmg/objecten?generatedAtTime=2023-08-12T00:01:27.217Z
             { actor: 'urn:comunica:default:rdf-resolve-hypermedia/actors#none' }

...
```

Code Fragment 2.4: (Cleaned up) logs outputted during execution of engine configured by files displayed in Code Fragments 2.2 and 2.3

```
{
  "@context": [
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        runner/^2.0.0/components/context.jsonld",
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        actor-extract-links-predicates/^0.0.0/components/context.jsonld"
  ],
  "@id": "urn:comunica:default:Runner",
  "@type": "Runner",
  "actors": [
    {
      "@id": "urn:comunica:default:extract-links/actors#predicates-common",
      "@type": "ActorExtractLinksPredicates",
      "checkSubject": false,
      "predicateRegexes": [
        "http://www.cidoc-crm.org/cidoc-crm/P129i_is_subject_of",
        "http://iiif.io/api/presentation/2#hasSequences",
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#first",
        "http://iiif.io/api/presentation/2#hasCanvases",
        "http://www.w3.org/2003/12/exif/ns#height",
        "http://iiif.io/api/presentation/2#hasImageAnnotations",
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#first",
        "http://www.w3.org/ns/oa#hasBody"
      ]
    }
  ]
}
```

Code Fragment 2.5: Comunica Predicates Extract Links Actor configuration with predicate regexes set to predicates from query displayed in Code Fragment 2.1 and subject checking **disabled**

The results immediately indicate that the Follow All engine struggles to execute the query successfully. It is important to note that the success rate is subject to a variety of factors, encompassing both client and server circumstances, such as the machine's specifications and the state of the internet connection. However, in this specific instance, the runtime error that emerged following unsuccessful link traversal was attributed to an excessive number of listeners assigned to a TLS socket. This situation may be associated with an overflow of HTTP requests. The combination of this issue with the absence of any valid results even after a considerable time span underscores that, for the objectives of this research, the Follow All engine, without additional configuration or the integration of supplementary actors, is unsuitable.

Fortunately, both the Follow Match Query and Custom engines were able to successfully execute their tasks. It is noteworthy, however, that the average times to resolve the query differ by only a few seconds. As expected, the custom engine performs better, but the marginal time saved initially might not seem significant compared to the drawback of having to adjust its configuration for each query. Nevertheless, it is reasonable to expect that the custom engine's advantage will become more pronounced when handling queries that target data distributed across multiple documents and situated at deeper levels. Moreover, it is entirely feasible to develop a user-friendly application that constructs the necessary configuration based on the specific query before executing the engine. This way, the configuration complexity can be abstracted from the end-users, providing a smoother user experience while harnessing the benefits of the custom engine's efficiency.

2.2.5 Traversing LDES Pages

Despite the custom engine's capacity to retrieve highly targeted data, its present form only accounts for a fraction of the available dataset. This limitation arises from the fact that an LDES comprises multiple pages, technically TREE nodes, necessitating both forward and backward *browsing* to encompass the entirety of the dataset. However, the predicates leading to the objects providing access to these other TREE nodes are presently absent from the regex array in the configuration of the Predicates Extract Links Actor.

While incorporating these predicates is straightforward, an even more effective approach involves introducing a second link extractor to the custom engine configuration. The *Extract Links Tree Actor*[12] possesses the capability to introduce links to the preceding and succeeding LDES *pages* - as identified by the *greater than* and *less than* relationships as defined within the TREE specification - into the link queue. This modest addition to the configuration profoundly enhances the capabilities of the resultant engine.

The revised configuration, as presented in Code Fragment 2.6, not only facilitates finely targeted searches for the requested data but also encompasses the complete dataset of the specified CoGhent institution(s) by leveraging the Extract Links Tree Actor.

2.2.6 Conclusion

In summary, an engine constructed using the Follow Match Query configuration, which utilizes the Quad Pattern Query Extract Links Actor, effectively addresses specific query requirements without necessitating additional actor configuration. However, for queries demanding more extensive traversal across documents or encompassing data distributed across mul-

[12] https://github.com/comunica/comunica-feature-link-traversal/tree/master/packages/actor-extract-links-extract-tree

```
{
  "@context": [
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        config-query-sparql/^2.0.0/components/context.jsonld",
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        config-query-sparql-link-traversal/^0.0.0/components/context.jsonld"
  ],
  "import": [
    "ccqslt:config/config-base.json",
    "./actors/extract-links-predicates-custom.json",
    "ccqslt:config/extract-links/actors/tree.json"
  ]
}
```

Code Fragment 2.6: Custom link traversal engine configuration using Predicates Extract Links Actor and Extract Links Tree Actor

tiple documents, a tailored configuration that integrates both the Predicates Extract Links Actor and the Extract Links Tree Actor can significantly enhance performance.

It is important to acknowledge that this approach does require a specific configuration outlining the predicates for each query. Nevertheless, this configuration complexity can be effectively abstracted from end-users through the development of tools that manage the technical intricacies behind the scenes. This approach ultimately strikes a balance between query performance optimization and user accessibility, aligning with the overarching goals of the research.

2.3 Links to Follow

Now that the data sources and engine to use have been determined, the focus can shift to creating queries. The exact data that these queries should retrieve is a choice left to the end-user. Chapter 3 delves deeper into the development of tools that can aid end-users in this process. However, before delving into that, this section first provides a closer look at various types of resources directly referenced from the CoGhent LDESs. These resources have the potential to generate interesting knowledge.

The types of resources discussed are as follows:

- CoGhent IIIF Manifests
- Wikidata
- Stad Gent (City of Ghent) data
- Getty Vocabularies

Unfortunately, some of these resources reveal certain technical limitations. These limitations are therefore also discussed, along with potential workarounds.

2.3.1 IIIF Manifest

As discussed in Section 1.4, each Human-Made Object within the CoGhent data links to a unique IIIF Manifest. An example of a link to such a CoGhent IIIF Manifest is https://api.collectie.gent/iiif/presentation/v2/manifest/dmg:3091. While the CoGhent LDESs contain descriptive data on Human-Made Objects, these CoGhent IIIF manifests specifically emphasize technical metadata for each Human-Made Object's digital *copy*. Typically, a CoGhent IIIF Manifest encompasses a single sequence, which in turn contains a single canvas, which further encapsulates an individual image.

The significance of image data in the cultural context of this research cannot be overstated. Or as the adage goes: *A picture is worth a thousand words*. In the realm of cultural heritage and art, images often convey intricate details, historical contexts, and artistic nuances that might be challenging to articulate through words alone.

Given the decision to employ the custom engine as discussed in Section 2.2, the queries are required to trace a sequence of predicates, initiating from Human-Made Objects and culminating in the desired object(s). However, the depth at which the valuable image data is nested within the manifest necessitates extensive predicate sequences, making the query notably lengthy. Due to this complexity, it might be tempting to gravitate towards the less stringent Follow Match engine, allowing queries to not necessarily contain long uninterrupted sequences of predicates. However, this approach can yield results erroneously associating Human-Made Objects with unrelated manifests.

To illustrate this hurdle, three hypothetical RDF documents are presented. They are hypothetical and are displayed in Turtle syntax in Code Fragments 2.7, 2.8 and 2.9. The first one depicts quads linking Human-Made Objects to their IIIF Manifests, and the other two respectively depict these two manifest. It must be noted that these examples in no way follow any of the schemas puth forth by CoGhent and IIIF. Notably, the IIIF Manifest examples deviate from the actual IIIF schema by eliminating the utilization of arrays. Instead, the examples assume a simplified scenario in which each manifest encompasses only one sequence, one canvas, and one image annotation.

```
ex:human_made_object1 ex:hasManifest ex:manifest1 .
ex:human_made_object2 ex:hasManifest ex:manifest2 .
```

Code Fragment 2.7: Turtle file representing hypothetical Human-Made Objects (does not follow CoGhent schema)

```
ex:manifest1 ex:firstSequence ex:sequence1 .
ex:sequence1 ex:firstCanvas ex:canvas1 .
ex:canvas1 ex:firstImageAnnotation ex:annotation1 .
ex:annotation1 iiif:hasBody ex:image1 .
```

Code Fragment 2.8: Turtle file representing **first** hypothetical IIIF Manifest (does not follow IIIF schema)

```
ex:manifest2 ex:firstSequence ex:sequence2 .
ex:sequence2 ex:firstCanvas ex:canvas2 .
ex:canvas2 ex:firstImageAnnotation ex:annotation2 .
ex:annotation2 iiif:hasBody ex:image2 .
```

Code Fragment 2.9: Turtle file representing **second** hypothetical IIIF Manifest (does not follow IIIF schema)

Naturally, the document containing the Human-Made Objects is designated as the initiation point for the link traversal process. If, and when, the chosen link traversal engine reaches the manifest links within this document and appends them to the link queue, the related manifest documents will subsequently be recognized and amalgamated with the previously identified document encompassing the Human-Made Objects. Code Fragment 2.10 presents the potential appearance of this amalgamation of documents.

```
# Human-Made Objects
ex:human_made_object1 ex:hasManifest ex:manifest1 .
ex:human_made_object2 ex:hasManifest ex:manifest2 .

# Manifest 1
ex:manifest1 ex:firstSequence ex:sequence1 .
ex:sequence1 ex:firstCanvas ex:canvas1 .
ex:canvas1 ex:firstImageAnnotation ex:annotation1 .
ex:annotation1 iiif:hasBody ex:image1 .

# Manifest 2
ex:manifest2 ex:firstSequence ex:sequence2 .
ex:sequence2 ex:firstCanvas ex:canvas2 .
ex:canvas2 ex:firstImageAnnotation ex:annotation2 .
ex:annotation2 iiif:hasBody ex:image2 .
```

Code Fragment 2.10: Turtle file representing combination of hypothetical Human-Made Objects and IIIF Manifests

Subsequently, two queries are introduced: a *long* query that meticulously delineates the path from a Human-Made Object to its image, as portrayed in Code Fragment 2.11, and a *short* query that seeks to streamline this process by eliminating the intermediary quad patterns, as displayed in Code Fragment 2.12. While these queries might appear, at first glance, to target the same data, the outcomes they yield are different. The outcomes of the *long* query are detailed in Table 2.7, whereas the outcomes of the *short* query are outlined in Table 2.8.

```
SELECT ?humanMadeObject ?image

WHERE {
    ?humanMadeObject ex:hasManifest ?manifest .
    ?manifest ex:firstSequence ?sequence .
    ?sequence ex:firstCanvas ?canvas .
    ?canvas ex:firstImageAnnotation ?annotation .
    ?annotation iiif:hasBody ?image .
}
```

Code Fragment 2.11: **Long** query fetching Human-Made Object and image

The inadequacy of the *short* query becomes apparent as it erroneously associates all conceivable Human-Made Objects with all potential images, unlike the accurate outcomes of the *long* query. Revisiting the amalgamation of documents presented in Code Fragment 2.10 and reevaluating the two queries provides insight into the root cause of the *short* query's shortfall.

```
SELECT ?humanMadeObject ?image

WHERE {
    ?humanMadeObject ex:hasManifest ?manifest .
    ?annotation iiif:hasBody ?image .
}
```

Code Fragment 2.12: **Short** query fetching Human-Made Object and image

Table 2.7: Results of **long** query displayed in Code Fragment 2.11 and RDF document displayed in Code Fragment 2.10

?humanMadeObject	?image
ex:human_made_object1	ex:image1
ex:human_made_object2	ex:image2

Table 2.8: Results of **short** query displayed in Code Fragment 2.12 and RDF document displayed in Code Fragment 2.10

?humanMadeObject	?image
ex:human_made_object1	ex:image1
ex:human_made_object1	ex:image2
ex:human_made_object2	ex:image1
ex:human_made_object2	ex:image2

Specifically, the *short* query neglects to establish a linkage between specific images and their corresponding Human-Made Objects. Conversely, the *long* query adeptly maintains this linkage by distinctly defining a *path* connecting Human-Made Objects to their associated images. Consequently, irrespective of the selected engine, queries should consistently formulate a well-defined trajectory from Human-Made Objects leading to the targeted objects.

Emphasizing the evident, this observation's relevance goes beyond queries exclusively targeting data within IIIF Manifests. Instead, this principle holds applicability across all types of queries and data sources that will continue to be explored within the scope of this research.

Guided by the aforementioned deliberations, one can construct working *document-overarching* queries - this time operating on real-world data - aimed at surveying the Human-Made Objects' IIIF Manifest data. For instance, Code Fragment 2.1 that was introduced at the beginning of this chapter, displays a query designed to extract the manifest URIs of ten specific Human-Made Objects, along with the corresponding height values and URIs leading to the associated image files within these manifests. It is noteworthy that the potential *drawback* stemming from extended queries due to lengthy predicate sequences is somewhat mitigated through the utilization of property path sequences.

2.3.2 Wikidata

Wikidata is a major player when it comes to RDF data. In simplified terms, Wikidata encompasses Wikipedia's key data points, but it presents them as structured Linked Data, adhering to RDF principles. Given that CoGhent's Human-Made Objects frequently reference Wikidata resources, this significantly opens the door to a wealth of additional knowledge. (van Veen, 2019)

While Wikidata as an organization seems to encourage users to primarily use their SPARQL endpoint, the data can also be retrieved in separate RDF documents. Furthermore, Wikidata operates a website[13] that enables very user-friendly and visual browsing through the data. However, here comes Wikidata's major pitfall: the resource and predicate URIs Wikidata uses for its SPARQL endpoint and website differ from the URIs the organization employs for its actual RDF data. In other words, to access the RDF documents, one needs to use URIs that Wikidata does not openly advertise. (Wikidata, 2023)

At first glance, this might seem to pose an issue for the link traversal process. After all, just like Wikidata advertises, the CoGhent data references the *standard* Wikidata URIs, not the RDF-specific ones. Fortunately, this does not disrupt the link traversal process, as these *standard* URIs are automatically resolved to their RDF-specific counterparts through content negotiation. For instance, an HTTP request asking for RDF data to the resource

`http://www.wikidata.org/entity/Q42`

is automatically redirected to

`https://www.wikidata.org/wiki/Special:EntityData/Q42`,

and a similar request to the predicate

`https://www.wikidata.org/wiki/Property:P17`

[13] https://www.wikidata.org/wiki/Wikidata:Main_Page

is automatically redirected to

`https://www.wikidata.org/wiki/Special:EntityData/P17`.

However, caution must be exercised when using Wikidata URIs in queries themselves. The link traversal engine being used is unaware of Wikidata's approach and thus will not be able to map quad patterns with *standard* Wikidata URIs in the query to quads with the RDF-specific URIs that appear in a fetched Wikidata RDF document. In other words, it is up to the user to *translate* the advertised Wikidata URIs into their RDF-specific counterparts. The application that controls the relevant link traversal engine could however also assist with this task.

Given these findings, queries can also be formulated to retrieve specific Wikidata information from Human-Made Objects. For instance, Code Fragment 2.13 presents such a query. Specifically, the query seeks to find the country where the cultural institution that possesses the particular Human-Made Object is located. Note that the Wikidata predicate URI indeed follows the same format as stored in the Wikidata RDF documents themselves.

```
PREFIX cidoc:<http://www.cidoc-crm.org/cidoc-crm/>
PREFIX wiki-prop:<http://www.wikidata.org/prop/direct/>

SELECT ?human_made_object ?country

WHERE {
  ?human_made_object cidoc:P50_has_current_keeper/wiki-prop:P17 ?country.
}

LIMIT 10
```

Code Fragment 2.13: SPARQL query fetching ten Human-Made Object's institute's countries

2.3.3 Stad Gent

The most common types of links in the CoGhent LDESs are arguably Stad Gent links. Stad Gent, which translates to *City of Ghent* in English, indeed also publishes a significant amount of its own data. This data often pertains to specific aspects of the city and might not be found in other thesauri. Since the city is inherently connected to CoGhent, its resources are certainly worth discussing.

Unfortunately, the Stad Gent links that provide context to the Human-Made Objects do not directly resolve to RDF documents. To illustrate this, two HTTP requests are executed, each targeting different Stad Gent URIs. Since these URIs might return various types of data depending on the request, the Accept value in the request header is explicitly set to "application/ld+json" each time.

Firstly, when running the command

```
curl -H "accept:application/ld+json"
     -iv "https://stad.gent/id/mensgemaaktobject/dmg/530005252/2023-08-12T00:01:27.217Z",
```

the server responds with an HTTP `406 Not Acceptable` response status along with the message *https://stad.gent/id/mensgemaaktobject/dmg/530005252/2023-08-12T00:01:27.217Z is not available in the requested format.* However, if the word `id` in the request URI is changed to `data`, the server responds successfully.

Secondly, running the command

```
curl -H "accept:application/ld+json"
     -iv "https://stad.gent/id/blank_node/bccb2bda-7563-4e94-82a4-ba8e9559d679"
```

results in an HTTP `404 Not Found` response status along with the message *no linked data representation of https://stad.gent/id/blank_node/bccb2bda-7563-4e94-82a4-ba8e9559d679 was found.* Fortunately, this issue can again be fixed by replacing `id` in the request URI with `data`.

While the server's treatment of distinct categories of Stad Gent URIs might not be immediately apparent, a single solution can be uniformly applied: substituting `id` with `data`. However, due to the CoGhent LDESs often referencing the types of CoGhent URIs that do not yield RDF data, the Comunica link traversal engine, attempting to request these URIs, will inevitably encounter the aforementioned error responses as well. Hence, the engine necessitates the capability to dynamically update any Stad Gent link that contains the string `id` before adding it to the link queue.

To achieve this, a new Comunica actor is built[14]. This actor is coined `ActorRdfResolveHypermediaLinksStadGentReplaceId`[15] and extends the `ActorRdfResolveHypermediaLinks` actor. The latter provides the new actor with access to the links that are being considered for addition to the link queue. Code Fragment 2.14 presents the `run` function of the new actor. Initially, the available links are iterated over, and using a regex, it is determined whether they match the pattern of a Stad Gent link containing an `id` path. Any link that meets this criteria is then modified by replacing the old path with a `data` path. At the end of the function, all the links, including the modified ones, are passed back to the bus allowing any subsequent actor to continue to work with them.

However, the actor also needs a way to indicate when its action has already been performed. If this is not done, the actor will be queried repeatedly, causing the engine to get stuck in an infinite loop. For this reason, just before concluding the `run` function, a key specific to the current action is set to `true`. This `KEY_CONTEXT_REPLACED` key then indicates during the actor's testing that the actor has already completed its task and should not be re-executed. The `test` function responsible for this behavior is depicted in Code Fragment 2.15.

After implementing the actor, it is given its own actor configuration, which is then imported into the custom engine configuration. This small addition ultimately enables an engine using it to involve Stad Gent resources in responding to a query. Or at least, that is the theory. In practice, however, Stad Gent documents are still not being identified. The reason for this is straightforward but unfortunate.

The Comunica engine executes its HTTP requests with a much broader `Accept` statement in the header compared to the one used in the manual `curl` tests from before. Code Fragment 2.16 shows exactly what this `Accept` statement looks

[14] Tutorial on building custom Comunica actor: https://comunica.dev/docs/modify/getting_started/contribute_actor/

[15] Implementation: https://github.com/thesis-Martijn-Bogaert-2022-2023/comunica-feature-link-traversal/blob/feature/change-gettyvocab-stadgent-links/packages/actor-rdf-resolve-hypermedia-links-stad-gent-replace-id/lib/ActorRdfResolveHypermediaLinksStadGentReplaceId.ts

```
public async run(action: IActionRdfResolveHypermediaLinks):
    Promise<IActorRdfResolveHypermediaLinksOutput> {
  const stadGentUriRegex = /^https?:\/\/stad\.gent\/id\/.+$/u;

  const links = action.metadata.traverse.map((link: { url: string }) => {
    if (this.stadGentUriRegex.test(link.url)) {
      const oldUrl = link.url;
      const newUrl = oldUrl.replace('/id/', '/data/');
      link.url = newUrl;
      this.logInfo(action.context, `Updated ${oldUrl} to ${newUrl}`);
    }
    return link;
  });

  // Update metadata in action
  const context = action.context.set(KEY_CONTEXT_REPLACED, true);
  const subAction = { ...action, context, metadata: { ...action.metadata, traverse: links }};

  // Forward updated metadata to next actor
  return this.mediatorRdfResolveHypermediaLinks.mediate(subAction);
}
```

Code Fragment 2.14: Implementation of `ActorRdfResolveHypermediaLinksStadGentReplaceId`'s `run` function

```
public async test(action: IActionRdfResolveHypermediaLinks): Promise<IActorTest> {
  if (action.context.get(KEY_CONTEXT_REPLACED)) {
    throw new Error('Already checked for Stad Gent links');
  }
  return true;
}
```

Code Fragment 2.15: Implementation of `ActorRdfResolveHypermediaLinksStadGentReplaceId`'s `test` function

like. And while RFC 7231[16] prescribes that `Content-Types` with higher quality values, denoted by `q`, indicate that they are preferred over lower ones, the Stad Gent server seems to ignore this and selects `text/html` as the `Content-Type`. Of course, this is not RDF data, meaning the Comunica engine cannot process it further. (Fielding and Reschke, 2014)

```
accept: 'application/n-quads,application/trig;q=0.95,application/ld+json;q=0.9,
         application/n-triples;q=0.8,text/turtle;q=0.6,application/rdf+xml;q=0.5,
         application/json;q=0.45,text/n3;q=0.35,application/xml;q=0.3,
         image/svg+xml;q=0.3,text/xml;q=0.3,text/html;q=0.2,
         application/xhtml+xml;q=0.18,text/shaclc;q=0.1,text/shaclc-ext;q=0.05'
```

Code Fragment 2.16: `Accept` header for HTTP requests made by Comunica engine

Making changes to the Comunica engine to remove `text/html` as an option is not feasible because it could negatively affect its overall functionality. Moreover, the issue clearly comes from the Stad Gent server's side. A simple adjustment on their end could potentially resolve the issue. But until that happens, the Stad Gent data, unfortunately, cannot be considered suitable for link traversal.

2.3.4 Getty Vocabularies

The last type of links found in the CoGhent LDESs are links to resources from Getty Vocabularies. On their official website, these resources are described as follows:

> *Getty Vocabularies are structured resources for the visual arts domain, including art, architecture, decorative arts, other cultural works, archival materials, visual surrogates, and art conservation. Compliant with international standards for structured and controlled vocabularies, they provide authoritative information for catalogers, researchers, and data providers.*
>
> (Getty Vocabularies, 2023)

In fact, The Getty Vocabularies encompass different thesauri. However, the one directly used by the CoGhent data, is called the *Art & Architecture Thesaurus*. Once again, the Getty Vocabularies website explains what this Thesaurus can be useful for:

> *The AAT includes generic terms, and associated dates, relationships, and other information about concepts related to or required to catalog, discover, and retrieve information about art, architecture, and other visual cultural heritage, including related disciplines dealing with visual works, such as archaeology and conservation, where the works are of the type collected by art museums and repositories for visual cultural heritage, or that are architecture.*
>
> (Art & Architecture Thesaurus, 2023)

It is clear that the Getty Vocabularies data, in combination with link traversal, can facilitate interesting and novel discoveries regarding Human-Made Objects. However, much like the previous resource providers, obtaining Getty Vocabularies data is not without its challenges. This is illustrated by the following quad pattern:

[16] https://datatracker.ietf.org/doc/html/rfc7231

```
?human_made_object
  cidoc:P41i_was_classified_by/cidoc:P42_assigned/<http://purl.org/dc/terms/created>
      ?created .
```

When attempting to execute a query with this quad pattern in the `WHERE` clause, no results are returned. To identify the issue, the first step is to retrieve only the URIs to the Getty Vocabularies documents. Note that due to the consequent truncation of the property path sequence, the query no longer needs to traverse to documents outside the CoGhent LDESs, therefore allowing a standard SPARQL engine to be used. Subsequently, one of the obtained URIs (e.g., http://vocab.getty.edu/aat/300037772) can be set as the data source for an engine with a simple task: retrieving the predicates and objects of those quads where the set data source is the subject. Consequently, this query yields the same situation as before: no results are retrieved.

When examining the query logs, as shown in Code Fragment 2.17, one thing stands out. One of the logs mentions a *missing context link header* and indicates the involvement of a document of type `application/json`. It is indeed rather surprising to see the Getty Vocabularies server return a JSON file. After all, Getty (2023) clearly states that *Data is delivered to a requesting agent through a standard triple serialization using HTTP RDF/XML, Notation-3 (N3), Turtle, N-Triples, RDFa, JSON, **JSON-LD***. This indicates that the server should be capable of offering JSON-LD documents, which in turn seems to indicate that the Getty Vocabularies server is not configured correctly either. After all, as illustrated in Code Fragment 2.16, the Comunica engine clearly indicates it prefers JSON-LD documents over JSON documents.

```
[...]  INFO: Requesting http://vocab.getty.edu/aat/300037772
       { ... , method: 'GET', actor: 'urn:comunica:default:http/actors#fetch' }
[...]  ERROR: Missing context link header for media type application/json
       on http://vocab.getty.edu/aat/300037772
       { actor: 'urn:comunica:default:dereference-rdf/actors#parse' }
[...]  INFO: Identified as file source: http://vocab.getty.edu/aat/300037772
       { actor: 'urn:comunica:default:rdf-resolve-hypermedia/actors#none' }
```

Code Fragment 2.17: (Cleaned up) logs outputted during execution of engine with data source set to Getty Vocabulary resource

However, it would be reasonable to assume that Comunica can handle JSON documents as long as they are valid RDF. To confirm this, the following command is executed:

```
curl -H "accept:application/ld+json" -iv "http://vocab.getty.edu/aat/300037772"
```

As observed before, the server returns a document with `Content-Type` of `application/json`. Yet, at first glance, this document appears to be a valid RDF document. This is confirmed by an RDF validator[17]. In addition, Getty Vocabularies resource documents can also be retrieved by appending one of the supported extensions to the *bare* resource URI. With that in mind, a final comparison can be made between the already obtained JSON content and the content that http://vocab.getty.edu/aat/300037772.jsonld leads to. This proves that both documents match word for word. The only difference lies in some

[17] https://www.w3.org/RDF/Validator/

special characters in one document being represented by their Unicode escape sequences, while in the other they appear in their literal form.

Nevertheless, whether the returned data has a `Content-Type` of `application/ld+json` or `application/json` should ideally not make a significant difference. After all, they both yield valid RDF content. However, to understand why the Comunica engine still seems to struggle with the JSON documents from the Getty Vocabularies server, a deeper dive into the log of the RDF Parse Actor mentioning *Missing context link header* is necessary.

To gain a better understanding of the engine's behavior, an examination of the tests[18] within the `ActorRdfParseJsonLd` actor is conducted. Even without delving into the implementation details, the *titles* of two tests offer insights. One test asserts that the actor *should run for a JSON doc with a context link header*, while the other asserts that the actor *should error on a JSON doc without a context link header*. The log in question aligns with what the latter test examines. Put simply, the Getty Vocabularies server provides its JSON content without a *context link header*, whereas the RDF Parse Actor expects such a header to be present. (Taelman et al., 2018)

This prompts two important questions: what is a *context link header*, and is the Comunica engine perhaps too strict? The answers can be found in W3's documentation[19]. Firstly, a context link header is a `Link` statement that can be included in the HTTP response header when returning JSON. This statement contains a URI that points to a JSON-LD context, enabling the JSON data to be interpreted as RDF data. Secondly, it can be confidently stated that Comunica rightfully expects a JSON response to be accompanied by such a context link header. Apart from using an *Alternate Document Location*, this is the only way to send JSON-LD syntax as JSON content. In other words, the behavior of the Comunica engine aligns perfectly with prescribed standards, while the behavior of the Getty Vocabularies server does not. (Sporny et al., 2020)

The Getty Vocabularies server's shortcomings lie in two aspects. Firstly, when the server receives a request explicitly asking for JSON-LD content, it should respond with JSON-LD content, especially considering that it indeed has such content available. Secondly, when the server receives a request specifically asking for JSON content, it should refrain from returning JSON with an `@context` property and instead provide a context link header in the response. This adherence to established conventions is essential for seamless interoperability between servers and clients in the RDF ecosystem.

Fortunately, there is still a way to salvage the Getty Vocabularies data without discarding it. As briefly mentioned before, there exists an alternative approach to explicitly instruct the Getty Vocabularies server to provide JSON-LD content. This involves adding a `.json-ld` extension to the URI in the request. However, to ensure clarity about the server's response behavior and to avoid any further unexpected outcomes, a small experiment is conducted. The experiment delves into two key aspects. First, it evaluates the effect of appending the `.json-ld` extension to the request URI on the returned content's `Content-Type`. Second, it investigates the potential influence of setting the `Accept` header to `application/ld+json` on the previous outcome. Finally, to ensure the findings are robust and not dependent on specific parameters like the queried Getty Vocabularies thesaurus or the type of resource (whether a *concept* or a *term*), the experiment is performed six times. Table 2.9 provides details about the queried URIs, the status of each request's `Accept` header, whether the `.json-ld` extension is used, and the `Content-Type` of the corresponding HTTP responses. The different abbreviations that appear in the *Thesaurus* column are *AAT*, *ULAN* and *TGN*, and respectively stand for *Art & Ar-*

[18] https://github.com/comunica/comunica/blob/master/packages/actor-rdf-parse-jsonld/test/ActorRdfParseJsonLd-test.ts
[19] https://www.w3.org/TR/json-ld/#interpreting-json-as-json-ld

chitecture Thesaurus, *Union List of Artist Names* and *Getty Thesaurus of Geographic Names*. Moreover, the six URIs defining the table's results, are http://vocab.getty.edu/aat/300043071, http://vocab.getty.edu/ulan/500115588, http://vocab.getty.edu/tgn/1000070, http://vocab.getty.edu/aat/term/1000043071-en, https://vocab.getty.edu/ulan/term/1500088448-en and https://vocab.getty.edu/tgn/term/26679-en, respectively.

Thesaurus	Type	JSON-LD extension	JSON-LD `Accept` header	`Content-Type`
AAT	Concept	✗	✗	`text/html`
		✗	✓	`application/json`
		✓	✗	`application/json`
		✓	✓	`application/ld+json`
ULAN	Concept	✗	✗	`text/html`
		✗	✓	`application/json`
		✓	✗	`application/json`
		✓	✓	`application/ld+json`
TGN	Concept	✗	✗	`text/html`
		✗	✓	`application/json`
		✓	✗	`application/json`
		✓	✓	`application/ld+json`
AAT	Term	✗	✗	`text/html`
		✗	✓	`application/ld+json`
		✓	✗	`404 Not Found`
		✓	✓	`application/ld+json`
ULAN	Term	✗	✗	`text/html`
		✗	✓	`application/ld+json`
		✓	✗	`404 Not Found`
		✓	✓	`application/ld+json`
TGN	Term	✗	✗	`text/html`
		✗	✓	`application/ld+json`
		✓	✗	`404 Not Found`
		✓	✓	`application/ld+json`

Table 2.9: Results from experiment examining `Content-Type`s of Getty Vocabularies server's HTTP responses

The results of the experiment show that the different thesauri are treated in the same way. Still, requests to *Concept* URIs and *Term* URIs appear to be handled differently by the Getty Vocabularies server. However, there is one particular combination that consistently returns `application/ld+json` content for every type of URI. This occurs when a request is sent that includes both a URI with the `.json-ld` extension and requests `application/ld+json` in the `Accept` header. Nevertheless, as mentioned several times, the Comunica engine makes HTTP requests with a much more extensive `Accept`

header. Fortunately, for the Getty Vocabularies server, this doesn't matter. As long as a `.json-ld` extension is used, the server returns actual JSON-LD data.

While all these findings are indeed interesting, they don't immediately solve the issue with a Comunica engine not being able to request actual JSON-LD data. For this reason, just as in Section 2.3.3, a new actor[20] is constructed that extends the `ActorRdfResolveHypermediaLinks` actor. Similarly to before, all available links are iterated through for potential adjustments. The implementation of the `run` function is shown in Code Fragment2.18, illustrating how the actor not only checks whether a link matches the template of a Getty Vocabularies resource URI, but also whether it doesn't already have an extension. If the link successfully passes both tests, it is eventually appended with a `.json-ld` extension. The `test` function, as shown in Code Fragment 2.19, functions similarly to previously, checking whether the actor has already been executed.

```
public async run(action: IActionRdfResolveHypermediaLinks):
    Promise<IActorRdfResolveHypermediaLinksOutput> {
  const gettyUriRegex = /^https?:\/\/vocab\.getty\.edu\/.+$/u;
  const extensions = [ '.json', '.jsonld', '.rdf', '.n3', '.ttl', '.nt' ];

  const links = action.metadata.traverse.map((link: { url: string }) => {
    if (this.gettyUriRegex.test(link.url)) {
      const hasExtension = this.extensions.some(ext => link.url.endsWith(ext));
      if (!hasExtension) {
        const oldUrl = link.url;
        const newUrl = `${oldUrl}.jsonld`;
        link.url = newUrl;
        this.logInfo(action.context, `Updated ${oldUrl} to ${newUrl}`);
      }
    }
    return link;
  });

  // Update metadata in action
  const context = action.context.set(KEY_CONTEXT_EXTENDED, true);
  const subAction = { ...action, context, metadata: { ...action.metadata, traverse: links }};

  // Forward updated metadata to next actor
  return this.mediatorRdfResolveHypermediaLinks.mediate(subAction);
}
```

Code Fragment 2.18: Implementation of `ActorRdfResolveHypermediaLinksGettyJsonldExtension`'s `run` function

[20] https://github.com/thesis-Martijn-Bogaert-2022-2023/comunica-feature-link-traversal/blob/feature/change-gettyvocab-stadgent-links/packages/actor-rdf-resolve-hypermedia-links-getty-jsonld-extension/lib/ActorRdfResolveHypermediaLinksGettyJsonldExtension.ts

```
public async test(action: IActionRdfResolveHypermediaLinks): Promise<IActorTest> {
  if (action.context.get(KEY_CONTEXT_EXTENDED)) {
    throw new Error('Already checked for Getty links');
  }
  return true;
}
```

Code Fragment 2.19: Implementation of `ActorRdfResolveHypermediaLinksGettyJsonldExtension`'s `test` function

To make this new actor operational, a dedicated configuration file is provided. This configuration file is then integrated into the custom engine configuration. Code Fragment 2.20 illustrates the configuration for the new actor, while Code Fragment 2.21 showcases the final configuration for the custom engine. This final configuration ultimately enables Comunica to build a link traversal engine that can not only precisely follow a query's predicate sequences and other LDES *pages* but also successfully incorporate Getty Vocabularies data into the query. This enhanced engine facilitates the execution of queries like the one proposed in Code Fragment 2.22. Notably, this query capitalizes on one of the most intriguing aspects of the Getty Vocabularies data: its extensive multilingual coverage. By integrating this data, the scope of the CoGhent dataset is significantly expanded, encompassing a diverse array of languages beyond just Dutch.

```
{
  "@context": [
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        runner/^2.0.0/components/context.jsonld",
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        actor-rdf-resolve-hypermedia-links-getty-jsonld-extension/^1.0.0/components/context.jsonld"
  ],
  "@id": "urn:comunica:default:Runner",
  "@type": "Runner",
  "actors": [
    {
      "@id": "urn:comunica:default:rdf-resolve-hypermedia-links/actors#getty-jsonld-extension",
      "@type": "ActorRdfResolveHypermediaLinksGettyJsonldExtension",
      "beforeActors":
        { "@id": "urn:comunica:default:rdf-resolve-hypermedia-links/actors#traverse" },
      "mediatorRdfResolveHypermediaLinks":
        { "@id": "urn:comunica:default:rdf-resolve-hypermedia-links/mediators#main" }
    }
  ]
}
```

Code Fragment 2.20: Extend Getty Links Actor configuration

```
{
  "@context": [
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        config-query-sparql/^2.0.0/components/context.jsonld",
    "https://linkedsoftwaredependencies.org/bundles/npm/@comunica/
        config-query-sparql-link-traversal/^0.0.0/components/context.jsonld"
  ],
  "import": [
    "ccqslt:config/config-base.json",
    "./actors/extract-links-predicates-custom.json",
    "ccqslt:config/extract-links/actors/tree.json",
    "./actors/rdf-resolve-hypermedia-links-traverse-extend-getty-links.json"
  ]
}
```

Code Fragment 2.21: Final custom link traversal engine configuration

```
PREFIX cidoc:<http://www.cidoc-crm.org/cidoc-crm/>
PREFIX skos-xl:<http://www.w3.org/2008/05/skos-xl#>
PREFIX getty:<http://vocab.getty.edu/ontology#>

SELECT *

WHERE {
  ?human_made_object cidoc:P41i_was_classified_by ?classifier.
  ?classifier cidoc:P42_assigned ?assignation.
  ?assignation skos-xl:prefLabel ?prefLabel.
  ?prefLabel getty:term ?thing.

  FILTER(LANG(?thing) = "de")
}

LIMIT 10
```

Code Fragment 2.22: SPARQL query fetching Human-Made Object's types in German

2.3.5 Conclusion

In this section, the four types of resources that arguably appear most frequently in the CoGhent LDESs were introduced and discussed. The challenges of accessing these resources through link traversal vary from type to type.

Firstly, the default link traversal functionality of Comunica has no problem reaching data within each Human-Made Object's IIIF Manifest. However, it is important to note that queries interrogating manifests, should explicitly navigate the complete (long) path from Human-Made Object to the object(s) of interest.

Next, accessing Wikidata resources also poses no issue for Comunica's default link traversal functionality. However, queries involving Wikidata URIs must match the RDF-specific URIs, not the *regular* ones advertised by Wikidata.

As for Stad Gent data, the responsibility lies with the Stad Gent server administrators. They need to adjust their server implementation to adhere to the prevailing standards so that Stad Gent resources can be successfully retrieved and parsed by a link traversal engine. At the time of publishing this research, this wasn't the case, making it impossible to involve the Stad Gent data in the link traversal process.

Lastly, the Getty Vocabularies server implementation also needs adjustment to conform to the established standards. However, to still enable a Comunica link traversal engine to query Getty Vocabularies documents, a temporary workaround can be used. This involves using a custom actor that explicitly requests valid JSON-LD content from the Getty Vocabularies server.

2.4 Conclusion

The investigation into CoGhent's data landscape, primarily focused on its characteristic Human-Made Objects, has brought to the forefront the pivotal role of link traversal in uncovering specific attributes of these objects. Expanding the scope of queries beyond the confines of CoGhent's internal data has opened up a wider range of insights and comparisons, thereby enabling unprecedented data exploration.

The distinctive Linked Data Event Streams (LDESs) associated with each institution within the CoGhent framework have emerged as crucial gateways for the Link Traversal-based Querying process. This distinction offers a nuanced approach, empowering users to selectively query information from individual institutions. However, it is important to acknowledge the inherent unpredictability of outcomes, inherent to LDESs and link traversal.

Leveraging the flexibility of the Comunica engine, a custom link traversal engine has been tailored to align with the distinctive demands of this research. Its configuration, which amalgamates the Predicates Extract Links Actor and the Extract Links Tree Actor, strikes a balance between enhancing query efficiency and ensuring user-friendly access.

Furthermore, the examination of resources directly linked from CoGhent's LDESs has illuminated potential domains rich in knowledge. While certain resources, such as CoGhent IIIF Manifests and Wikidata, are readily accessible, others like Stad Gent data and Getty Vocabularies present certain challenges. Nonetheless, proactive solutions have been identified, offering partial and temporary avenues to navigate these challenges.

Building on the foundation laid in this chapter, Chapter 3 will introduce tools designed to assist individuals without a technical background in formulating queries. Rooted in the principles discussed during the past chapter, these tools will generate

queries optimized for a link traversal engine, constructed based on the specified custom configuration. This approach aims to seamlessly bridge the technical intricacies with user convenience, thereby ensuring an enriched and accessible user experience.

3

Tools for Query Building

Discovering digital art collections encompasses a wide array of possibilities, each with its unique interpretations and implementations. This research, however, primarily centers on the facilitation aspect of this discovery process. The CoGhent collections undoubtedly harbor a treasure trove of potentially captivating insights, yet professionals and art enthusiasts can only unlock these treasures if they can formulate the right SPARQL queries. This task is far from simple, particularly when considering that these individuals might lack the technical proficiency required to construct such queries. Consequently, this chapter introduces and partially develops two conceptual web applications designed to significantly alleviate the technical complexities of query formulation for users.

The first application draws inspiration from the existing CoGhent Query Builder proposed in Section 1.4.3. The fundamental concept remains unchanged: users are presented with a list of properties, and based on their selections, the application constructs a query with the necessary triple patterns to retrieve the desired data. However, the *enhanced* iteration of this application takes two further strides. Firstly, it introduces modularity by ensuring that the properties and their corresponding triple patterns are not hard-coded into the application. Instead, they are provided as sequences of predicates through JSON files. Secondly, the application supports the generation of *cross-dataset queries* that can be effectively resolved through a link traversal engine, as elaborated in Chapter 2.

The second application targets users with a slightly more advanced understanding of RDF. It empowers them to explore the predicate sequences of properties of interest themselves. This exploration journey begins with an RDF resource provided by the user. From this initial *root* resource, users can progressively construct a tree comprising of predicates and objects. Based on users' choices, the application executes queries to fetch the predicates and objects associated with an already-present resource in the tree. As users gain a deeper understanding of the data accessible through the given root resource, they can select specific objects within the tree. The application then deduces the predicate sequences leading from the root resource to these chosen objects and subsequently generates a corresponding query. Crucially, since this query solely relies on predicate sequences, it empowers users to perform a generalized inquiry on their entire dataset(s), not just the resource specified at the beginning of the process.

Both of these applications are briefly introduced in Sections 3.2 and 3.3, respectively, by discussing their main features and most essential implementation details. For the complete implementations, readers are referred to their respective GitHub repositories: https://github.com/thesis-Martijn-Bogaert-2022-2023/sparql-query-builder-ui and https://github.com/thesis-Martijn-Bogaert-2022-2023/rdf-predicates-explorer. However, prior to this, Section 3.1 first covers the fundamental func-

tionality shared by both web applications. As to avoid repeating code and to ensure separation of concerns, this functionality lives on its own and is incorporated into the other two applications by importing its implementation as a package. The detailed implementation of this essential tool can be found in the following GitHub repository: https://github.com/thesis-Martijn-Bogaert-2022-2023/sparql-query-builder.

3.1 Building Queries from Predicate Sequences

To provide user-oriented applications with the capability to obtain the corresponding query based on an input of predicate sequences, this section introduces a Node.js application that accomplishes precisely that. The application does not have a user interface; it solely exports a function that performs the described functionality. This allows other Node.js applications to use the function by installing the application as a package.

The actual implementation can be found in the following GitHub repository:

https://github.com/thesis-Martijn-Bogaert-2022-2023/sparql-query-builder.

3.1.1 Arrays of Triple Patterns

The simplest but somewhat naive way to build queries in the application is by maintaining the necessary triple patterns for each *property* they represent in a single string. Based on the properties selected by the user, the application can then place the corresponding strings one by one in the `WHERE` clause of a new query. However, starting from such completely hard-coded *chunks* of triple patterns not at all meets the requirement of working with *predicate sequences*, makes it difficult to *clean up* the query, and is simply no elegant solution.

The next logical approach is to keep an array of strings for each property, with each string representing a single triple pattern. Code Fragment 3.1 provides an example of what such an array might look like. Note that the array itself effectively becomes the value of a key-value pair, the key being the property name. This allows to *feed* the function responsible for the actual query building with a dictionary of such key-value pairs.

```
objectname: [
    '?s cidoc:P41i_was_classified_by ?classified.',
    '?classified cidoc:P42_assigned ?assigned.',
    '?assigned skos:prefLabel ?objectname.',
]
```

Code Fragment 3.1: WHERE clause statements to query for *objectname* stored as elements in an array

An important and convenient feature of SPARQL is the ability to use prefixes. Code Fragment 3.1, for instance, assumes that the provided triple patterns are already utilizing prefixes. Naturally, in this case, the application should construct a query that begins with the necessary `prefix` statements. The simplest, yet again naive, way to achieve this is by including the same set of `prefix` statements at the start of each query. Code Fragment 3.2 illustrates what this set of `prefix` statements might look like.

```
PREFIX cidoc:<http://www.cidoc-crm.org/cidoc-crm/>
PREFIX adms:<http://www.w3.org/ns/adms#>
PREFIX skos:<http://www.w3.org/2004/02/skos/core#>
PREFIX la:<https://linked.art/ns/terms/>
```

Code Fragment 3.2: All possible PREFIX statements of the original CoGhent Query Builder

It hardly needs mentioning that the solutions described above come with several issues. First, breaking down the *chunks* of triple patterns into separate array members still does not allow for a clean query generation process. Second, hard-coded prefixes hinder the modularity of the app. After all, only accepting triple statements with prefixes from such a limited predefined list is hardly user-friendly. Moreover, even if the app aims to accommodate as many types of prefixes as possible, the question arises: how extensive should that list - and thus each query - become?

3.1.2 Arrays of Predicates

To address both of these shortcomings, an improved approach is proposed. It is important to realize that due to the nature of the queries being formulated in this research, only the predicate of each triple pattern is of real importance. The subjects and objects are consistently variables, and storing their names along with the predicate does not serve much purpose. Therefore, the elements in each property's array do not need to correspond directly to explicit triple patterns, but solely to predicates. This really brings the concept of *predicate sequences* to the forefront. Additionally, to have a better understanding of which prefixes are potentially used, these can also be extracted from the predicate strings and stored separately. With these observations in mind, Code Fragment 3.3 introduces a new data structure. In it, each property to be included in the query, holds an array of objects. In turn, each of these objects includes a mandatory predicate field and, in case the latter is no URI, a prefix field as well.

```
objectname: [
    { prefix: 'cidoc', predicate: 'P41i_was_classified_by' },
    { prefix: 'cidoc', predicate: 'P42_assigned' },
    { prefix: 'skos', predicate: 'prefLabel' },
]
```

Code Fragment 3.3: Prefixes and predicates for WHERE clause statements to query for *objectname* stored as elements in an array

Immediately, it becomes apparent that using this new data structure is much more elegant than the initially proposed approach. However, the downside is that the query building function now has more work to do. Since variable names are no longer specified, the function needs to generate them. This can be approached in two ways.

Firstly, the function could use a generic variable name like `var` and append an ever increasing number to it. The major drawback of this solution, though, is that generic variable names can make the query less readable. For instance, while Code Fragment 3.4 required introducing *only* four such variable names, it becomes evident that a larger number of them would make the final query less user-friendly.

```
# Objectname
?var1 cidoc:P41i_was_classified_by ?var2.
?var2 cidoc:P42_assigned ?var3.
?var3 skos:prefLabel ?var4.
```

Code Fragment 3.4: WHERE clause statements with object variable names constructed using numbers

As a second approach, the function could use variable names determined by the preceding variable names and predicates. Code Fragment 3.5 demonstrates that this type of variable naming indeed clearly indicates their purpose. However, in terms of user-friendliness, this approach scores even worse than the previous one. Queries quickly become overloaded with overly long variable names, rendering them barely understandable and certainly not readable.

```
# Objectname
?s
    cidoc:P41i_was_classified_by
        ?s_cidoc_P41i_was_classified_by.
?s_cidoc_P41i_was_classified_by
    cidoc:P42_assigned
        ?s_cidoc_P41i_was_classified_by_cidoc_P42_assigned.
?s_cidoc_P41i_was_classified_by_cidoc_P42_assigned
    skos:prefLabel
        ?s_cidoc_P41i_was_classified_by_cidoc_P42_assigned_skos_prefLabel.
```

Code Fragment 3.5: WHERE clause statements with object variable names constructed from preceding statements

It is evident that none of the approaches mentioned above is the optimal solution. Therefore, in Section 3.1.3, things are being approached differently one last time. However, before proceeding, another dilemma needs to be addressed. Namely, when multiple properties are involved in a query, it is entirely possible that parts of their *paths* toward their respective end objects, overlap. In other words, it is plausible that the query function needs to add the same triple pattern to the `WHERE` clause multiple times. Code Fragment 3.6 illustrates what such a query might look like. Indeed, in principle, the second occurrence of the triple pattern `?o cidoc:P108i_was_produced_by ?produced` is redundant. After all, once a SPARQL engine reaches this triple pattern, it will utilize the existing bindings for the variables `?o` and `?produces`, rendering this triple pattern dispensable.

```
# Place
?o cidoc:P108i_was_produced_by ?produced.
?produced cidoc:P7_took_place_at ?tookplace.
?tookplace la:equivalent ?plaatsequivalent.
?plaatsequivalent skos:prefLabel ?plaats.
# Date
?o cidoc:P108i_was_produced_by ?produced.
?produced cidoc:P4_has_time-span ?timespan.
```

Code Fragment 3.6: WHERE clause statements with overlapping statements

Code Fragment 3.7 depicts the same query as displayed in Code Fragment 3.6, but without duplicate triple patterns. Such a query is indeed more compact, but the query-building process becomes slightly more complicated. For instance, to keep track of the various predicates already used, along with their subject and object variables, the entered *flat* predicates dictionary could be transformed into a tree structure, where branches represent unique predicates and nodes denote intermediary variable names. This approach would subsequently allow for building queries void of duplicate triple patterns. However, one might question whether it is worthwhile to implement such *complex* logic. In fact, even though permitting duplicate triple patterns might potentially lengthen the resulting queries, it simultaneously contributes to more comprehensible and lucid queries. This is evidenced by the query in Code Fragment 3.6: it is readily apparent, both for the *place* property and the *data* property, which *paths* must be traversed to retrieve their respective objects of interest. Bearing this in mind, this research prioritizes the creation of these *clear* queries.

```
# Place
?o cidoc:P108i_was_produced_by ?produced.
?produced cidoc:P7_took_place_at ?tookplace.
?tookplace la:equivalent ?plaatsequivalent.
?plaatsequivalent skos:prefLabel ?plaats.
# Date
?produced cidoc:P4_has_time-span ?timespan.
```

Code Fragment 3.7: WHERE clause statements without overlapping statements

3.1.3 User-Defined Variable Names and Property Path Sequences

As discussed in the previous section, a way must be devised to handle the usage of variable names. Since this research aims to assist individuals without the necessary technical knowledge in comprehending too complex queries, two additional functionalities are introduced. These functionalities aim to both condense queries and make them more intelligible.

To fulfill the first objective, property path sequences are employed. These sequences essentially concatenate consecutive predicates, eliminating the need for variable names. To achieve the second objective, users are provided with the option to add an `object_variable_name` key to the triple pattern descriptions in the properties dictionary. For instance, when the query-building function encounters an `object_variable_name`, it will not concatenate the next predicate to the current one using a property path sequence. Instead, it will use the specified variable name for the object of the current triple pattern, as well as for the subject of the next one. Additionally, a `subject_variable_name` can also be included. However, to avoid clashes with potential `object_variable_name`s, this will only be respected for the first triple pattern description in an array. This should provide the capability to deviate the starting point of a sequence of triple patterns from the *default* starting point.

In principle, the functionality described above should suffice for building clear, albeit very simple queries. For that reason, before discussing a handful additional features in Sections 3.1.4 and 3.1.5 Code Fragments 3.8 and 3.9 are introduced. Code Fragment 3.8 presents two dictionaries intended to be passed to the query-building function. The `properties` dictionary attempts to illustrate the concepts discussed earlier. Specifically, the dictionary outlines *paths* to several objects of interest. For the `title` and `description` properties, only one predicate is needed each, while the `objectname` and `association` properties require three and four predicates, respectively. For each element in the property arrays, a

`predicate` is provided; logically, this is the only mandatory named value. Additionally, prefixes are added where necessary. The query-building function will place these prefixes before their corresponding predicates. To ensure a functional query, however, it is assumed that the URIs representing the prefixes are provided separately to the query-building function. The latter will subsequently use them to create `PREFIX` statements at the beginning of the query. Moving on, no beginning array element is assigned a `subject_variable_name`, which should result in a query where each *path* starts from the same subject variable. However, two `object_variable_name` specifications are provided. Their corresponding variable names should appear in the resulting query, essentially *breaking* the regular property path sequences. Indeed, as depicted in Code Fragment 3.9, the output of the query-building function matches the expectations perfectly.

```
const properties = {
    title: [
        { prefix: 'cidoc', predicate: 'P102_has_title' }
    ],
    description: [
        { prefix: 'cidoc', predicate: 'P3_has_note', object_variable_name: 'note' },
    ],
    objectname: [
        { prefix: 'cidoc', predicate: 'P41i_was_classified_by' },
        { prefix: 'cidoc', predicate: 'P42_assigned' },
        { predicate: 'http://www.w3.org/2004/02/skos/core#prefLabel' },
    ],
    association: [
        { prefix: 'cidoc', predicate: 'P128_carries' },
        { prefix: 'cidoc', predicate: 'P129_is_about', object_variable_name: 'about' },
        { prefix: 'cidoc', predicate: 'P2_has_type' },
        { predicate: 'http://www.w3.org/2004/02/skos/core#prefLabel' },
    ],
};

const prefixes = {
    cidoc: 'http://www.cidoc-crm.org/cidoc-crm/',
};
```

Code Fragment 3.8: Properties and prefixes ready to be consumed by query building function

3.1.4 Filtered and Optional Properties

As with the original CoGhent Query Builder, this application should provide the ability to filter and/or make properties optional. While, technically, a SPARQL query can have these specifications defined anywhere in its WHERE clause, in the orginal CoGhent Query Builder's and this application's case, these specifications are intended to be defined per *property*. This only makes sense, as from the perspective of simplicity and consistency, both applications aim to abstract the complexities of query building to a large extent. Therefore, it might be inappropriate to provide users with the ability to manipulate the

```
PREFIX cidoc:<http://www.cidoc-crm.org/cidoc-crm/>
PREFIX skos:<http://www.w3.org/2004/02/skos/core#>

SELECT ?title ?note ?objectname ?association

WHERE {
    # title
    ?human_made_object cidoc:P102_has_title ?title.

    # description
    ?human_made_object cidoc:P3_has_note ?note.

    # objectname
    ?human_made_object
        cidoc:P41i_was_classified_by/cidoc:P42_assigned/skos:prefLabel
            ?objectname.

    # association
    ?human_made_object cidoc:P128_carries/cidoc:P129_is_about ?about.
    ?about cidoc:P2_has_type/skos:prefLabel ?association.
}
```

Code Fragment 3.9: SPARQL query generated from input displayed in Code Fragment 3.8

abstracted triple patterns at the beginning. Besides, once the query is generated, users can obviously still modify it as they wish.

To include the filtering functionality, one approach is to provide a new dictionary to the query-building function where property names can be listed along with their filter details. However, introducing a third dictionary might become a bit cluttered for developers. Therefore, an alternative approach is to incorporate the filter details into the existing `properties` array. For this, a slight modification to the data structure is needed. Code Fragment 3.10 demonstrates what this entails. Specifically, the array containing predicate information is no longer the direct value of its predicate key. Instead, the value of the predicate key becomes a dictionary that has room for a `statements` key, which in turn houses the original array of statements.

```
const properties = {
    title: {
        statements: [
            { prefix: 'cidoc', predicate: 'P102_has_title' }
        ],
    },
    description: {
        statements: [
            { prefix: 'cidoc', predicate: 'P3_has_note', object_variable_name: 'note' },
        ],
        filters: { string: 'luchter', language: 'nl' },
        optional: true,
    },
};
```

Code Fragment 3.10: Example of properties dictionary to illustrate use of filters and optionals

Now that each property in the properties array can host various types of data, there is finally space for a `filters` key. This key specifies a new dictionary where multiple types of filters can be accommodated. In fact, unlike the original CoGhent Query Builder, this opens up the possibility of allowing different kinds of filters alongside a regex-based and case-insensitive `string` filter. Of course, this is provided that the query-building function knows how to handle them. At the time of publishing this research, an additional feature has been implemented which allows specifying a `language` filter value. Code Fragment 3.11 presents the corresponding query based on the properties dictionary in Code Fragment 3.10 and demonstrates how the language filter is reflected in the query.

Code Fragments 3.10 and 3.11 also illustrate how a property can be made optional. This can be achieved simply by adding the `optional` key to the property details and setting its value to `true`.

3.1.5 Limit and Offset

Finally, the application is also capable of adding a `LIMIT` and/or `OFFSET` statement to the query. Their corresponding parameters, `limit` and `offset`, can simply be passed as parameters to the query-building function. However, it is worth noting that, in the context of a link traversal engine, an offset might not be very meaningful due to the unpredictable nature

```
PREFIX cidoc:<http://www.cidoc-crm.org/cidoc-crm/>

SELECT ?title ?note

WHERE {
    # title
    ?human_made_object cidoc:P102_has_title ?title.

    # description
    OPTIONAL {
        ?human_made_object cidoc:P3_has_note ?note.

        FILTER(REGEX(?note, "luchter", "i"))
        FILTER(LANG(?note) = "nl")
    }
}
```

Code Fragment 3.11: SPARQL query generated from input displayed in Code Fragment 3.10

of the order in which results are discovered, as discussed in Section 2.1.2. Nonetheless, in case a user really desires to see different results, adjusting the offset might increase the likelihood of retrieving previously unseen results. On the other hand, providing a limit is very much advantageous. Since link traversal can be slow and theoretically even infinite, a limit partly alleviates these issues by instructing the query engine to stop gathering links and results after a certain count.

3.1.6 Overview

The application is clearly highly powerful when it comes to constructing simple queries, specifically queries that target specific data points. Creating input for the query-building function is less tedious than manually composing the corresponding query. Nevertheless, throughout the previous sections, numerous *rules* have been presented, making it appropriate to summarize them. It is important, however, to understand that the use, and therefore the preparation of input for the query-building function, is not intended for the end user. On the contrary, the described application is designed to be incorporated by other applications that ultimately - and hopefully - provide the end user with a user-friendly user interface. In other words, the developers of these end applications are the ones who need to know how to properly provide the query-building function with the right parameters.

Specifically, the query-building, or in fact `buildQuery`[1] function can be provided with five parameters. The following overview discusses them in order:

1. **properties** (required)
   This should be a dictionary with each key indicating a property. In turn, each property specifies a dictionary containing the following named values:

[1] https://github.com/thesis-Martijn-Bogaert-2022-2023/sparql-query-builder/blob/main/index.js

- **statements** (required)
  This should be an array containing one or more dictionary elements. The order of the elements decides the *path* to follow. Each element contains the following named values:
  - **predicate** (required)
    Specifies the predicate as a string. This can be a full URI or only the end of one, thus expecting a prefix.
  - **prefix** (optional)
    Specifies the prefix as a string. Should only be used if the predicate expects a prefix.
  - **subject_variable_name** (optional)
    Explicitly sets the subject variable name of the corresponding triple pattern and will solely be handled upon in case it is part of an array's first element. In case an array's first element does not have this specified, the subject variable name will be set to `?o`.
  - **object_variable_name** (optional)
    Explicitly sets the object variable name of the corresponding triple pattern, as well as the subject variable name of the subsequent triple pattern. In case an array's last element does not have this specified, the object variable name will be set to the property key. In case an array's any other element does not have this specified, the current and subsequent predicates will be concatenated using a property path sequence.
- **filters** (optional)
  This should be a dictionary containing the following names values:
  - **string** (optional)
    Specifies the string to filter this property's last triple pattern's object name on as a string.
  - **language** (optional)
    Specifies the language to filter this property's last triple pattern's object name on as a string.
- **optional** (optional)
  Specifies whether or not to make the retrieval of this property optional as a boolean.

2. **prefixes** (optional)
   This should be a dictionary that specifies which `PREFIX` statements to add to the start of the query. Each key represents the prefix name, while each value represents the corresponding URI.

3. **datasets**[2] (optional)
   This should be an array of string elements. Each element specifies the URI of a specific graph to query against and will be mapped to the value of a `FROM` statement in the query

[2] This functionality was not discussed since the various CoGhent LDESs already inherently partition the entire CoGhent dataset. Therefore, the use of `FROM` statements for the type of queries addressed in this research is not relevant.

4. **limit** (optional)
   This should be an integer and will be mapped to the query's `LIMIT` statement.

5. **offset** (optional)
   This should be an integer and will be mapped to the query's `OFFSET` statement.

3.2 A Modular Query Builder

The application introduced in this section closely mirrors the functionality of the original CoGhent Query Builder. However, due to its reliance on the query-building function as outlined in Section 3.1, the resultant queries possess the potential to target a broad spectrum of datasets beyond just CoGhent's.

There are two other substantial differences with the original CoGhent Query Builder as well. Firstly, the application's modularity is evident in the sense that the presented properties are not rigidly embedded within the application's codebase. Instead, they are dynamically retrieved from distinct JSON files. Secondly, the generated queries have the capacity to traverse beyond single documents, necessitating execution through a link traversal engine.

Just like the query builder application introduced in Section 3.1, this section too introduces a Node.js application. However, this one now boasts a user interface (UI). Nonetheless, the UI is so user-friendly and intuitive that these technical intricacies do not find coverage within this research. Simply put, users are initially presented with an overview of available *modules*. They can then *open* these modules and make selections from the displayed properties. With each modification of this selection, the application updates the corresponding query. It is worth noting that while the original CoGhent Query Builder generates its queries only when a button is clicked, this application aims to provide users with a better understanding of the query-building process by translating their actions into results in real-time.

The actual implementation can be found in the following GitHub repository:

https://github.com/thesis-Martijn-Bogaert-2022-2023/sparql-query-builder-ui.

3.2.1 Modularity

As announced, this application introduces a form of modularity by offering property selection based on independent JSON files. On the one hand, the application looks into the `config/` directory at the root of the project, and on the other, users can upload their own JSON files. The key property of such JSON files is, of course, its `properties` key. It aligns perfectly with the type of properties dictionary that needs to be passed to the query-building function from Section 3.1. Its schema thus corresponds entirely to the schema outlined in Section 3.1.6. When forwarding the data of the user-selected properties to the query-building function, the application subsequently merely needs to parse the selected JSON properties into JavaScript objects and aggregate them together in a JavaScript dictionary. This dictionary is then ready to be passed as the first parameter of the query-building function.

In addition to the `properties` key, a JSON file can optionally include a `prefixes` key. This key specifies a dictionary that maps prefixes to their URIs. When the selected properties are passed to the query-building function, the application will

check for any used prefixes in their `statements` arrays. Using this information, the application can retrieve the necessary prefix definitions from the `prefixes` JSON dictionary and pass them to the query-building function.

Modularity is a useful feature, but it's important to consider that the targeted `config/` directory can potentially contain a large number of JSON files. To prevent the application from performing too many and potentially unnecessary I/O operations on start-up, the application initially refrains from loading the contents of the JSON files. Instead, it reads all the JSON file names and uses them to provide users with an overview of the available *modules*. Only when a user decides to *expand* a module, the contents of the corresponding JSON file are loaded, and its properties are presented to the user as selectable *blocks*.

It must be acknowledged that working with these *bare* JSON files does not entirely align with the spirit of Linked Data. The use of such independent resources could indeed prompt their publication in RDF format. However, since this part of the research mainly focuses on the query-building process, this functionality has not been developed. Nonetheless, if the need arises in the future, this is a direction that can certainly be investigated more extensively.

3.2.2 Signifying Intent with Questions

As a reminder, the UI of the application essentially displays a list of selectable properties. Each property is represented by a *block* displaying the property's name and providing the option to select the property. What hasn't been mentioned yet is that these *blocks* also provide filtering options. Specifically, a user can provide a string filter and select a language from the corresponding dropdown list.

While many of these UI elements are also present in the original CoGhent Query Builder, this application introduces an additional component that could potentially make the query-building process more comprehensible and powerful for users. Namely, for each property in a JSON file, a `question` key can be included. This should give an indication of the kind of question that can be answered by selecting the respective property. Of course, these questions are not passed to the query-building function; they are only used to be displayed in each property *block* and assist the user in constructing queries.

3.3 Discovering Predicate Sequences

The previous tool, while making query construction very accessible for absolute beginners, does have a clear drawback: the queries that can be created depend on the already available *properties*. In other words, users rely on the work of others. To cater to users who are a bit more adventurous and want access to the *theoretically* complete list of properties without requiring them to have prior knowledge of the schema of the data source they're querying, an additional tool is introduced in this section.

This application allows users to gain a better understanding of how the data in their dataset is structured before selecting properties of interest. This is achieved by letting users provide a specific resource URI that should indicate the kind of predicates and objects that can be reached by starting from essentially any such resource. In other words, this *starting* resource serves as a blueprint for the schema of all its *colleague* resources. For instance, consider the CoGhent LDESs; they encompass a plethora of Human-Made Objects. If a user wants to query the CoGhent LDESs but has no idea about the kind of data that Human-Made Objects grant access to, they can provide the URI of any Human-Made Object as the starting point for

the application. The application then allows the user to freely explore all branches and sub-branches departing from this resource. Armed with that knowledge, the user can subsequently select specific data points that sound interesting to them. The query builder application from Section 3.1 can then generate a query from the corresponding *predicate paths*, which can be executed across the entire CoGhent LDESs, using all available Human-Made Objects as starting points.

It must be acknowledged, however, that the system of solely relying on the user to specify the starting resource's URI goes somewhat against the goal of *user-friendliness*. After all, to get this URI, the user is essentially expected to have queried their dataset before - at least partially. While not implemented in the code provided with the research, several ways can be brought into place to alleviate this issue. For instance, the application could potentially ask the user to initially only provide access to their dataset. From that dataset, the application could then extract the resource URIs itself - one per *type* - and let the user choose on of them as the starting point. Additionally, a very simple search system could even be integrated to help the user find an even more *thought out* starting point.

The actual implementation can be found in the following GitHub repository:

https://github.com/thesis-Martijn-Bogaert-2022-2023/rdf-predicates-explorer.

3.3.1 Tree Data Structure and Visualization

As illustrated by Figure 1.2, a web of Linked Data can typically be represented using a graph. In this representation, nodes represent resources or atomic values, and edges represent predicates. For the development of the application central to this section, relying on a graph structure is therefore a good option. However, in the interest of clarity for users and to avoid unnecessarily complicating the development process of the application, the choice is made to avoid working with cycles. In other words, the discovery of new predicates and resources is supported by a tree structure. After all, opting for a tree data structure not only enhances the visualization aspect of the application but also aids in storing the data.

Developing a tree data structure itself is no insurmountable task. However, developing a tree's visualization aspect is less straightforward. Therefore, existing systems are leveraged. Given that the application is a Node.js application, there are numerous libraries that can be considered. For instance, a very popular choice for any kind of data visualization is *D3.js*[3]. However, while this library offers a lot of possibilities, achieving a user-friendly tree interface still requires substantial implementation work. Therefore, a better approach would be to seek out libraries specifically focused on tree visualization. One such library is *visjs-network*[4][5]. This library does indeed make it remarkably simple to provide nodes and edges with custom data and neatly visualize the entire tree, including pan and zoom functionality. However, during its use within the context of this research, it became evident that this package still presents challenges regarding visualization customization. After all, there needs to be enough space in the nodes and at the edges to accommodate the corresponding resource and predicate URIs. A more fitting alternative is therefore *cytoscape.js*[6]. This library too makes tree data management exceedingly simple, but it also allows for a fairly straightforward arrangement of the tree's design so that even custom data can be made legible.

[3] https://www.npmjs.com/package/d3

[4] https://www.npmjs.com/package/visjs-network

[5] The *visjs-network* library is a fork of the now depricated *vis.js* library

[6] https://www.npmjs.com/package/cytoscape

Figure 3.1 displays a screenshot of the application, illustrating the final tree design.

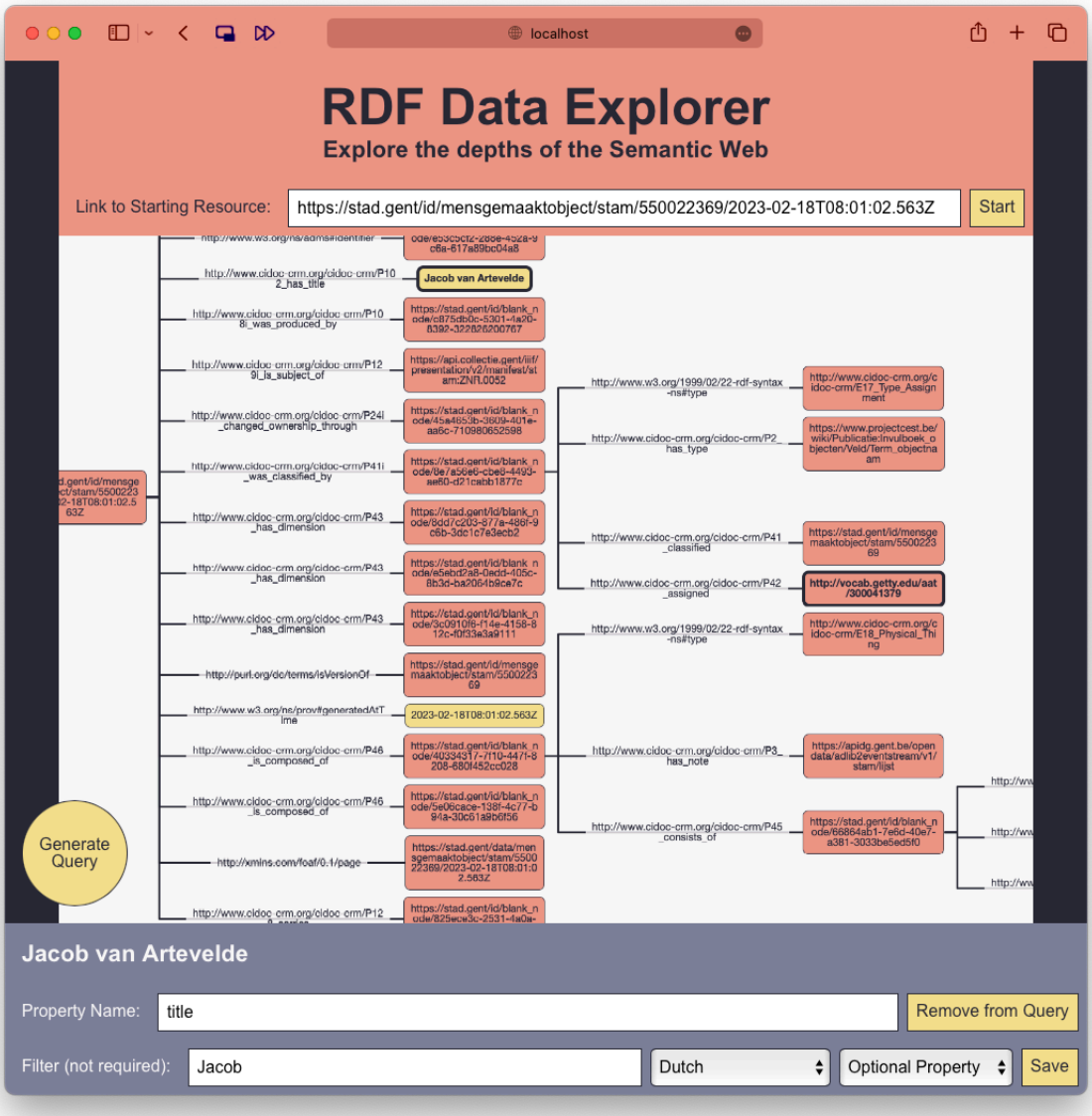

Figure 3.1: Screenshot of RDF Predicates Explorer

3.3.2 Tree Expansion

Another advantage to using *cytoscape.js* is the ease with which event listeners for node clicks can be added. This is necessary as users need to interact with nodes. For instance, if a node represents a resource, not an atomic value, users should be able to *expand* it. This involves retrieving the predicate and object for every triple pattern in which the resource is the subject. Consequently, for each such predicate-object pair found, a corresponding edge-node pair is added to the resource node. This

system allows users to build a tree of predicates and resources, providing insights into the part of the web of Linked Data that the starting resource grants access to.

Naturally, behind this *expanding* operation lies a SPARQL query. That query is not at all complicated, as exemplified by Code Fragment 3.12. Here, the function that returns the necessary query for a given resource URI is presented. To execute the query, the application uses Comunica's standard SPARQL engine[7], rather than a link traversal engine. The latter isn't necessary for this type of query, as there is no need for a link traversal engine that might needlessly search for potentially *followable* links and carries some overhead anyways. Furthermore, the used query engine naturally requires a datasource. Logically, the resource URI should suffice for this. At least, that is the theory. Because in practice and as attested to in Section 2.3, certain resource URIs - for various reasons - do not always lead directly to a *queryable* document. For instance, Getty Vocabularies URIs only return correct JSON-LD content when a `.json-ld` extension is appended to them. Therefore, to also enable users to expand these *affected* resources, the application provides the possibility to modify the datasource before executing the query from Code Fragment 3.12. As a bonus, this also enables users to specify a SPARQL endpoint as the datasource, potentially speeding up querying.

```
function buildQuery(subjectResource) {
    return `
        SELECT ?p ?o
        WHERE {
            <${subjectResource}> ?p ?o.
        }
    `;
}
```

Code Fragment 3.12: Function returning a SPARQL query for completing a resource subject's triple pattern

3.3.3 Predicate Sequences Selection

The purpose of the application central to this section is, of course, to compose queries. To achieve this, this application also utilizes the query builder application discussed in Section 3.1. The query builder function of the latter expects several parameters, with the most important being the `properties` parameter. In other words, for each type of property the user wants to include in their query, at least the corresponding *predicate sequence*, and optionally some filter details and an `optional` value, must be provided. Compiling all of this into a valid `properties` dictionary is a trivial task for the application. However, to understand the user's preferences, the necessary UI elements need to be provided. Code Fragment 3.1 already provides a visual indication of those.

Specifically, the application presents an input form when a node is clicked. Initially, it only offers the option to enter a property name. The intention is for this to briefly describe the type of data obtained by following the predicate sequence to the node in question - whether a resource or an atomic value. Subsequently, once the property name is provided, the user can add the property to the query. This step presents them with a few additional yet optional fields. Specifically, the user can provide a string filter - the label of the node in question is automatically offered as an option - choose a language, and designate the property as required or optional. Each node included as a property in the query is highlighted in the tree.

[7] https://github.com/comunica/comunica/tree/master/engines/query-sparql

Consequently, when the user is satisfied with their choices, they can press the *Generate Query* button. Upon this signal, the application first collects all properties with their respective details into a dictionary, structured according to the rules defined in Section 3.1.6. For each chosen node, this also requires the various predicates leading from the starting resource to that node. Fortunately, *cytoscape.js* can help with this too. After all, for each node, it offers the `predecessors()`[8] function. This leaves the application only with filtering each selected node's predecessors for just edges - predicates - and reversing their order. Eventually, once the complete `predicates` dictionary is determined, the application passes it to the query-building function, which ultimately presents the generated query to the user. As an added bonus, the application can even convert the dictionary to a JSON file, allowing it to be used as a *module* in the application discussed in Section 3.2.

3.4 Conclusion

The applications presented in Sections 3.2 and 3.3 are primarily user-centric. The application in Section 3.2, for instance, serves as a great starting point for absolute beginners to get a high-level idea of certain datasets, as well as how the selection of specific properties, accompanied by questions, translates into the construction of a SPARQL query. However, the drawback is that users rely on existing *modules* tailored to specific datasets. Without these modules, the application has no use. This is why the application discussed in Section 3.3 was developed. It allows users to select properties based on the branches and sub-branches of a specific resource in their dataset. However, this application expects a bit more technical understanding from users. Not only do users need to provide the URI of the starting resource themselves, *expanding* the tree can also be a somewhat tedious process.

Certainly, both applications have their distinct strengths and weaknesses. However, the key functionality of both, which is query building, is performed by a separate application. This application provides a query-building function that enables developers to build various user-centric applications *around* it. The only prerequisite is to provide the function with the appropriate parameters. Once again, the specific details of these parameters are set out in Section 3.1.6.

In any case, the ultimate goal of each application discussed in this chapter is the creation of a query. The logical next step is for a user to execute this query. However, given that the generated queries are *document-transcending*, users need an appropriate link traversal engine to execute them. The custom engine developed at the end of Chapter 2 serves this purpose. However, it is important to note that improperly configured servers might react unexpectedly to requests from such engines, and there is thus no guarantee that every query will yield results. Temporary workarounds can however be implemented to handle such cases. For instance, the custom engine from Chapter 2 can work with Getty Vocabularies resources thanks to a newly-created actor. At the time of publication, this research therefore recommends using this custom engine.

Nevertheless, the question now becomes where users are expected to consult these engines. One option would be to manually build a standalone application *around* the engine, another to simply host a *Comunica SPARQL jQuery Widget*[9] with the necessary configuration. However, to save users from unnecessary copying and pasting, this research has chosen to expand the functionality of the applications from Sections 3.2 and 3.3. Specifically, when a query is generated in either application, users are provided with the option to execute it immediately.

[8] https://js.cytoscape.org/#nodes.predecessors

[9] https://github.com/comunica/jQuery-Widget.js/

This leaves only one last question to answer: what to do with these query results? Chapter 4 delves into exploring potential resolutions to the challenge of handling those.

4

Handling Query Results

Chapter 3 introduced tools for formulating queries, and Chapter 2 covered executing these queries using a link traversal engine. Yet, certain questions remain unanswered. This chapter addresses these concerns.

Firstly, it deals with a crucial aspect pertinent to the core datasets of this study: digital art collections. Given the significance of visual data in these collections, the challenge of visualizing query results, arises. This is explored in Section 4.1.

Secondly, a more universal issue is addressed; the need for saving query results for future reference. How can this be effectively achieved? This *preservation* issue is discussed in Section 4.2.

It is important to clarify that this chapter does not strive to offer exhaustive analyses of these topics. Instead, it provides an overview of potential solutions, outlining their advantages and drawbacks. Moreover, no single solution is deemed inherently superior to the other(s).

4.1 Visualizing Query Results

Given the research's focus on retrieving data as properties of specific CoGhent Human-Made Objects, the visual aspect of these objects revolves around displaying their corresponding digital images. The fact that each object is associated with a single image simplifies matters. Additionally, the provision to also showcase textual data for each object is crucial. Code Fragment 2.1 serves as an example of a query that acquires some textual attributes for every Human-Made Object, alongside the object's digital image URI.

Sections 4.1.1 and 4.1.2 delve into the advantages and disadvantages of two visualization approaches. Irrespective of the method chosen, a key requirement is the ability to map the query results into the tool's internal structure. This entails that the visualization tools cannot possibly accommodate any query results without user instructions. They either solely accept results that strictly adherence to a predefined schema, or they offer a mapping interface empowering users to define how and where specific properties should be displayed.

4.1.1 IIIF Viewers

As discussed in Section 1.5, IIIF Viewers are commonly used tools for visualizing cultural data. Therefore, they are also suitable candidates for visualizing query results that may arise from this research. The greatest advantage of using IIIF Viewers is, of

course, that they don't need to be developed from scratch. Users have a wide range to choose from. Moreover, since these viewers are generally open-source projects, users can even customize an existing IIIF Viewer to their liking.

Using a IIIF Viewer also implies that the query results need to be mapped to a IIIF Manifest. However, since these manifests can be structured in various ways, there can be different approaches to this mapping. Certain decisions need to be made, such as which Presentation API version the manifest should support - whereas Presentation API `3.*` is the latest and most capable version, some IIIF Viewers only support Presentation API `2.*` - as well as how the results should be organized. In the context of this research, a proof-of-concept mapper was developed, supporting Presentation API `2.*` and providing one canvas per Human-Made Object, all grouped together in a single sequence.

The actual implementation can be found in the following GitHub repository:

https://github.com/thesis-Martijn-Bogaert-2022-2023/iiif-generator.

While using an existing IIIF Viewer indeed eliminates a lot of implementation work, it has one major drawback. IIIF Viewers expect the IIIF Manifest they are supposed to visualize, to be provided via its resource URI. While this aligns with the Linked Data principles, it restricts a true *discovery* process. Since IIIF Manifests must be hosted - whether locally or externally - the IIIF Viewer cannot dynamically update itself when new results prompt changes in the corresponding IIIF Manifest. Hence, if such *live update* feature holds significance, alternative solutions must be explored.

4.1.2 Custom Viewer

To have a true real-time visualizer at their disposal, developers need to take matters into their own hands. Fortunately, they can still rely on existing open-source tools. For instance, developers can work with the *Annona Library*[1]. This library offers a somewhat more *makeshift* IIIF Viewer, allowing the flexibility of presenting a IIIF Manifest as a string and deliberately abstaining from being classified as an *official* IIIF Viewer. In principle, extending such a viewer to react in real-time to the results of a query engine should be achievable.

However, relying solely on existing tools might impose restrictions on certain customization aspects. Consequently, developing a custom viewer from the ground up is also a valid approach. Nevertheless, the considerable implementation effort involved in this approach must be thoughtfully evaluated.

4.2 Saving Query Results

The query results acquired through this research might uncover new insights into the employed datasource(s). This naturally raises the need to archive these findings for future reference. There are various methods to achieve this, each approaching the notion of *results* from a distinct angle.

Initially, as discussed in Section 4.2.1, results can be retained through direct storage - that is, by saving the corresponding *bindings objects* for each *bindings*. This approach ensures that the results remain accessible at any point. However, it also introduces the risk of the retained data not being up-to-date with the original data anymore. After all, the original data might

[1] https://github.com/ncsu-libraries/annona

have undergone changes or even been removed since the last retention. Moreover, while retaining query results literally, can be beneficial and deliberate, it could also present legal considerations. For instance, in case the copyright information for a CoGhent Man-Made Object's image is updated, this change will not be reflected in the stored results.

In contrast, Sections 4.2.2 and 4.2.3 adopt a fundamentally different perspective on the concept of *results*. Here, the focus is not so much on the specific query outcomes but rather on the instructions to reproduce them. Consequently, in both cases, these instructions themselves are viewed as the data worthy of retention. While this approach implies that users don't possess specific results at their fingertips, it guarantees that upon executing these instructions, the obtained results consistently align with the current state of the utilized datasource(s).

4.2.1 IIIF Manifest

The most straightforward way to store query results literally is by using a text-based file format. Examples include `.csv` and `.txt` files. Using a custom database is also an option, albeit a somewhat more advanced one. Alternatively, a IIIF Manifest can be used, identical to what was discussed in Section 4.1.1. However, in this scenario, the query results must be accurately integrated into the relevant sections of the manifest. Still, this approach offers the advantage of immediate and perpetual visualization of the results.

4.2.2 SPARQL Query

To retain the instructions for acquiring query results, a direct approach is to store the actual SPARQL query. This can be accomplished by saving it in a `.rq` file. However, when retrieving the query later on, it is essential to determine which query engine should execute it. This is due to the fact that certain queries are closely tied to the engine they were tailored to. This holds true for the type of queries central to this study as well. Notably, many of these queries are tailored to the custom engine introduced in Section 2.2 and may not function correctly when executed by others.

4.2.3 Predicate Sequences

For queries specifically targeting Human-Made Objects in CoGhent's collections, an alternative method exists to preserve the instructions leading to query results. Section 3.2 namely introduced a JSON data structure to maintain corresponding *predicate sequences* for various properties. The associated query builder application subsequently translates these JSON files into executable queries. One benefit of this approach is that it allows for the files to be shared as *modules* with other users. However, this storage method is arguably quite niche, and users who are unfamiliar with the application may struggle to interpret these JSON files. Therefore, if the retention of query instructions is a significant consideration, it is advisable to store them directly as SPARQL queries.

4.3 Conclusion

This chapter has delved into the intricate nuances of handling query results, building upon the foundational knowledge established in the preceding chapters on formulating and executing queries. The emphasis on digital art collections, particularly the CoGhent Human-Made Objects, has underscored the importance of visualizing and preserving query results.

The visualization of query results, as discussed in Section 4.1, given the visual nature of the datasets, poses unique challenges. While existing tools like IIIF Viewers offer a ready-made solution, they come with their own set of limitations, particularly in terms of real-time updates. On the other hand, custom viewers, though demanding in terms of development, offer greater flexibility and real-time capabilities.

The preservation of query results, as elaborated in Section 4.2, presents a *dilemma*: retaining the actual results versus preserving the instructions to reproduce them. While direct storage methods, such as text-based files or IIIF Manifests, offer immediate access to results, they risk becoming outdated or misaligned with the original data. In contrast, storing the SPARQL query or predicate sequences ensures that results are always up-to-date with the current state of the data source, albeit at the cost of immediate accessibility.

In essence, the handling of query results is a multifaceted process, with each method offering its own set of advantages and drawbacks. The choice largely depends on the specific requirements and constraints of the user or project. While the tools and methods presented in this chapter provide a comprehensive overview of the possibilities, it is essential to approach the handling of query results with a clear understanding of the desired outcome and the limitations of each method.

Conclusion

The exploration of digital art collections using Link-Traversal-based Query Processing has been a multifaceted journey, intertwining the realms of art and technology. As digital art collections have become more accessible, challenges have arisen, particularly for individuals without a technical background. This research embarked on addressing these challenges, aiming to provide tools and methodologies that empower both professionals and art enthusiasts to delve deeper into the digital art landscape, making new discoveries and drawing meaningful connections.

The systematic process of discovering digital art collections can be broadly categorized into three main steps: building queries, executing them —with a specific focus on link traversal in this research —and handling the results through visualization and storage. Each chapter of this research has meticulously addressed one of these steps.

Chapter 2 delved into the execution of queries using Link-Traversal-based Query Processing. The chapter emphasized the potential of link traversal in uncovering specific attributes of the Collections of Ghent's Human-Made Objects, offering insights that would remain hidden with traditional querying methods. However, it also highlighted the inherent challenges and unpredictability associated with link traversal. For instance, due to server misconfigurations, certain resources like Stad Gent's could not be accessed, while the use of others', like Getty Vocabularies', required workarounds. These challenges underscored the fragility of Link Traversal-based Query Processing compared to traditional SPARQL querying.

Chapter 3 introduced tools that simplify the intricate task of query formulation, making it more accessible to a broader audience. While these tools are valuable, they are not presented as the definitive solutions. Instead, their core query-building functionality is designed to be modular, allowing others to adapt and use it for their own discovery applications.

Chapter 4 addressed the post-query phase, focusing on the visualization and preservation of query results. The chapter weighed the advantages and disadvantages of different visualization and preservation methods. However, it also highlighted areas for further exploration, such as the adaptability of visualization tools to real-time query updates and the potential for presenting query results as immersive, interactive narratives. These areas present exciting avenues for future research.

In conclusion, this research has provided valuable insights and tools for the discovery process of digital art collections, while also shedding light on the inherent challenges of the process. Specifically, the fragility of link traversal, its slower performance compared to traditional SPARQL querying, and the unpredictability of outcomes due to server misconfigurations or other unforeseen technical issues, pose significant hurdles. However, the undeniable potential of link traversal to uncover hidden data and offer deeper insights into digital art collections offers a promising future. As technology evolves and these challenges are addressed, it is anticipated that link traversal and its associated tools will become more mainstream, benefiting a wider audience. Yet, it is important to note that, at the time of this research's publication, harnessing its full capabilities still demands a certain level of technical expertise.

Ethical and social reflection

TODO

References

Art & Architecture Thesaurus (2023). About the aat. https://www.getty.edu/research/tools/vocabularies/aat/about.html.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2007). Dbpedia: A nucleus for a web of open data. In *The Semantic Web: 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007+ ASWC 2007, Busan, Korea, November 11-15, 2007. Proceedings*, pages 722–735. Springer.

Beckett, D. (2014). RDF 1.1 n-triples. W3C recommendation, W3C. https://www.w3.org/TR/2014/REC-n-triples-20140225/.

Beckett, D., Berners-Lee, T., Prud'hommeaux, E., and Carothers, G. (2014). RDF 1.1 turtle. W3C recommendation, W3C. https://www.w3.org/TR/2014/REC-turtle-20140225/.

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

Berners-Lee, T. and Connolly, D. (2011). Notation3 (n3): A readable rdf syntax. W3C team submission, W3C. http://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/.

Bizer, C., Heath, T., and Berners-Lee, T. (2011). Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI global.

Buil-Aranda, C., Corby, O., Das, S., Feigenbaum, L., Gearon, P., Glimm, B., Harris, S., Hawke, S., Herman, I., Humfrey, N., Michaelis, N., Ogbuji, C., Perry, M., Passant, A., Polleres, A., Prud'hommeaux, E., Seaborne, A., and Williams, G. T. (2013). SPARQL 1.1 overview. W3C recommendation, W3C. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/.

Candan, K. S., Liu, H., and Suvarna, R. (2001). Resource description framework: metadata and its applications. *Acm Sigkdd Explorations Newsletter*, 3(1):6–19.

CoGhent (2022). Linked data event streams. https://coghent.github.io/apiendpoints.html.

Dongo, I. and Chbeir, R. (2019). S-RDF: A New RDF Serialization Format for Better Storage Without Losing Human Readability. In *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*, 28th International Conference on COOPERATIVE INFORMATION SYSTEMS, pages 246–264, Rhodes, Greece. Springer International Publishing.

DuCharme, B. (2013). *Learning SPARQL: querying and updating with SPARQL 1.1.* " O'Reilly Media, Inc.". http://www.snee.com/semwebmeetup/2011-09-15/SPARQLBobDuCharme.pdf.

Fielding, R. T. and Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231.

Gandon, F., Schreiber, G., and Becket, D. (2014). RDF 1.1 XML syntax. W3C recommendation, W3C. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

Getty (2023). Getty vocabularies and linked open data (lod). https://www.getty.edu/research/tools/vocabularies/Linked_Data_Getty_Vocabularies.pdf.

Getty Vocabularies (2023). Getty vocabularies. https://www.getty.edu/research/tools/vocabularies/index.html.

4 References

Golbeck, J. and Rothstein, M. (2008). Linking social networks on the web with foaf: A semantic web case study. In *AAAI*, volume 8, pages 1138–1143.

Jacksi, K. and Abass, S. M. (2019). Development history of the world wide web. *Int. J. Sci. Technol. Res*, 8(9):75–79.

MDN Web Docs (2023). 302 found. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302.

Miller, E. (1998). An introduction to the resource description framework. ディジタル図書館, 13:3–11.

Powers, S. (2003). *Practical RDF: solving problems with the resource description framework*. O'Reilly Media, Inc.

Seaborne, A. and Harris, S. (2013). SPARQL 1.1 query language. W3C recommendation, W3C. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Champin, P.-A., and Lindström, N. (2020). JSON-ld 1.1. W3C recommendation, W3C. https://www.w3.org/TR/2020/REC-json-ld11-20200716/.

Taelman, R. (2020). Quad pattern fragments. W3C draft, W3C. https://linkeddatafragments.org/specification/quad-pattern-fragments/.

Taelman, R., Van Herwegen, J., Vander Sande, M., and Verborgh, R. (2018). Comunica: a modular sparql query engine for the web. In *Proceedings of the 17th International Semantic Web Conference*.

Van de Vyvere, B., D' Huynslager, O. V., Atauil, A., Segers, M., Van Campe, L., Vandekeybus, N., Teugels, S., Saenko, A., Pauwels, P.-J., and Colpaert, P. (2022). Publishing cultural heritage collections of ghent with linked data event streams. In *Metadata and Semantic Research: 15th International Conference, MTSR 2021, Virtual Event, November 29–December 3, 2021, Revised Selected Papers*, pages 357–369. Springer.

van Veen, T. (2019). Wikidata: From "an" identifier to "the" identifier. *Information Technology and Libraries*, 38:72–81.

Wikidata (2023). Wikidata:data access. https://www.wikidata.org/wiki/Wikidata:Data_access.

Appendices

Appendix A

TODO

Appendix B

TODO