JSON-LD Representations of CityJSON

Auteur : Abdelaleem, Aly
Promoteur(s) : Debruyne, Christophe
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2023-2024
URI/URL : https://github.com/aly1551995/CityJSON-LD/; http://hdl.handle.net/2268.2/21151

UNIVERSITY OF LIÈGE

FACULTY OF APPLIED SCIENCES

SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

JSON-LD Representations of CityJSON

*Master's thesis completed in order to obtain the degree of
Master of Science in Computer Science*

Author

ABDELALEEM Aly

supervisor

Prof. DEBRUYNE Christophe

Academic year 2023-2024

Abstract

CityJSON is a JSON encoding for 3D city models that provides a lightweight, easy-to-read, and developer-friendly substitute for the traditional CityGML format. However, the increasing demand for semantic interoperability and linked data necessitates using JSON-LD because it allows the incremental addition of semantics to existing JSON documents. This thesis focuses on using JSON-LD to represent CityJSON data to bridge the gap between city model data and Web 3.0 (Semantic Web). We take a closer look at how to convert the CityJSON format into a more semantically rich format and convert the implicitness of CityJSON into a more explicit format that allows for easily sharing and querying data across multiple and different systems. We will also highlight the changes JSON-LD could bring to managing urban data by creating a more connected and semantically interoperable approach to modeling cities.

Acknowledgments

I want to extend my deepest gratitude to my supervisor, Professor Christophe Debruyne, for his constant help and the encouragement that he always provided throughout my thesis. His expertise and knowledge have been invaluable in this academic pursuit.

I would also like to thank my family and friends for their constant encouragement and unshakable support during this thesis.

Disclaimer

The text of this thesis has not been generated using text-generative AI tools; however, tools such as Grammarly and QuillBot were used for grammar checking and applying formal writing conventions, additionally ChatGPT as a coding assistance tool.

Contents

Acronyms

AI Artificial Intelligence.

CJ2JLD CityJSON To JSON-LD.

CJIO CityJSON/io.

CLI Command Line Interface.

CRS Coordinate Reference System.

GEOS Geometry Engine - Open Source.

GIS Geographic Information System.

GML Geography Markup Language.

GUI Graphical User Interface.

HDT Header Dictionary Triples.

HTML Hypertext Markup Language.

INSPIRE Infrastructure for Spatial Information in the European Community.

IRI Internationalized Resource Identifier.

ISO International Standards Organization.

JSON JavaScript Object Notation.

JSON-LD JavaScript Object Notation for Linked Data.

LOD Level Of Detail.

OGC Open GIS Consortium.

OWL Web Ontology Language.

RDF Resource Description Framework.

RDFS Resource Description Framework Schema.

SDI Spatial Data Infrastructure.

SHACL Shapes Constraint Language.

SPARQL SPARQL Protocol and RDF Query Language.

SQL Structured Query Language.

UML Unified Modeling Language.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

W3C World Wide Web Consortium.

WKT Well-known Text (WKT).

WWW World Wide Web.

XML Extensible Markup Language.

1 Introduction

The popularity and complexity of 3D urban data led to the need for a more effective and standard way of encoding such data. Even though the currently available formats provide such services, the current formats still have caveats. For instance, CityGML is considered too verbose and complex; however, CityJSON is compact and utilizes JSON-based encoding for 3D city urban models [1]. However, both formats still lack the semantic richness required by new applications, which incorporate the Semantic Web principles that enable data integration and interoperability in smart cities. This thesis aims to bridge the gap by converting CityJSON to a more semantically rich format that is also JSON-based, called JSON-LD, enabling improvements in the semantic capabilities of 3D city models. Making it easier to share data, integrate data, and do advanced queries.

1.1 Research Question

Our main problem is finding an effective way to convert the CityJSON format to JSON-LD to enhance semantic interoperability and integrate 3D city model data. This problem leads us to our primary research question:

How can we effectively convert CityJSON into JSON-LD to enhance semantic interoperability and facilitate the integration of 3D urban data?

1.2 Research Objectives

The primary goal of this thesis is to identify a "JSON-LD representation of CityJSON." By converting the developer-friendly CityJSON format to the machine-friendly JSON-LD format, we leverage the advantages of both technologies. To achieve this, we must meet the following objectives:

- Understanding CityJSON and other 3D city modeling formats and their limitations will help us better comprehend the need for JSON-LD for semantic interoperability, as shown in Chapters 2 and 3.

- The development of a methodology to convert CityJSON to JSON-LD while maintaining and extending its semantics, as showcased in Chapter 4.

- The implementation of the previous approach takes the form of a CityJSON to JSON-LD conversion tool, as explored in Chapter 5.

- Evaluate the conversion tool's performance and effectiveness in real-world scenarios, as documented in Chapters 6, 7, and 8.

1.3 Thesis Structure

The thesis is structured as follows:

- **Chapter 1: Introduction**
  This chapter outlines the context, the research question, the objective, and the structure of the thesis.

- **Chapter 2: Background**
  This chapter summarizes essential information to enhance readers' comprehension of the thesis topic.

- **Chapter 3: Literature**
  This chapter explores some literature reviews and related work.

- **Chapter 4: Approach**
  This chapter comprehensively describes the approach taken to convert CityJSON to JSON-LD.

- **Chapter 5: Implementation**
  This chapter showcases the technical aspects of the conversion process.

- **Chapter 6: Demonstration**
  This chapter provides a brief demonstration with dummy data.

- **Chapter 7: Case Study**
  This chapter lists the tool's applicable case studies using CityJSON data of real cities such as Helsinki and New York City.

- **Chapter 8: Discussion**
  This chapter discusses the challenges, the limitations, and potential areas of improvement.

- **Chapter 9: Conclusion**
  This final chapter summarizes the work done throughout the thesis and explores potential future work.

2   Background

This chapter provides a background for the fundamental concepts underlying this thesis: the Semantic Web and the role of Spatial Data Infrastructure (SDI) in managing geospatial data.

2.1   Semantic Web

First, a brief history of the World Wide Web (WWW) is needed to understand the big picture. The first iteration, Web 1.0, or the static Web, marked the revolutionary beginnings of the Web. During this phase, individuals with minimal computer proficiency could access information simply by clicking hyperlinks. Web 1.0 relied on Hypertext Markup Language (HTML) to create static pages with hyperlinks for navigating between different documents. The problem with Web 1.0 is that it was static and unidirectional, meaning publishing content required significant experience and capabilities, which led to the emergence of the next generation of the World Wide Web (WWW). Web 2.0, also known as the Social Media Web, was more interactive because users could publish and interact with it in real time. It also featured the use of JavaScript, giving the HTML pages a more dynamic feel [2]. The rise in popularity of artificial intelligence (AI) is one of the features that spurred the emergence of a new Web propelled by machine learning. Such a Web is called a Semantic Web, or Web 3.0, characterized by the use of AI to ingest users' data, which is easily accessed now due to the previous era of Web 2.0, to customize the Web experience and target it based on the user's behavior [3]. Web 3.0 enables machines to understand the content of the HTML page, which improves the user's experience by inferring the correct choice just from the semantics and context used by the user, hence the name Semantic Web. When a user searches for the word "apple" on Google, the user's browser might analyze the previous searches and personal data to determine the user's intent. As a result, it could prioritize showing the user's search results related to the new iPhone or the fruit, depending on the user's browsing history [2].

Standards and technologies are required to allow machines to understand the semantics and syntax of data on the Internet. The following sections explain the key components that allow machines to process and interpret information meaningfully, allowing for a better and more helpful internet experience.

2.1.1   Ontology

According to the Oxford Dictionary, *ontology* is "the science or study of being; that branch of metaphysics concerned with the nature or essence of being or existence" [4]. In the Semantic Web context, according to [5], an *Ontology* is "A formal, explicit specification of a shared conceptualization." The definition refers to **conceptualization**, which is an abstraction of an object; it also refers to **explicit** (The documentation of all concepts in a non-ambiguous way in an external document that machines can access), **formal** (using a formalism so that one can use reasoners), and **shared** (by shared meaning to allow access by multiple systems to ensure the common understanding of the domain, meaning there is consensus about the ontology); it describes concepts, relations, and properties in a structured and standardized way without obscurities or misunderstandings [5]. Moreover, the Resource Description Framework (RDF) is a formal, machine-readable way to conceptualize such ontologies. RDF provides a set of vocabulary that allows the

modeling of real-life objects in a machine-friendly manner. Machines can interrupt and reason with this model to represent knowledge, achieve semantic interoperability, and make logical inferences on the data based on the ontology's axioms. Ontologies facilitate extensibility by serving as templates for creating new ontologies and promoting reusability [6]. RDF is an abstract graph data model that relies on IRIs to name resources, allowing one to create a distributed graph. RDF models data as triples: subject, predicate, and object, as shown in Figures 1 and 2.
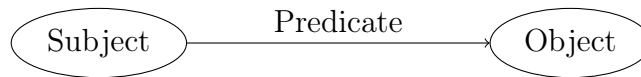

Figure 1: RDF illustration

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix ex: <http://example.org/>.

ex:Person1 rdf:type foaf:Person;
    foaf:name "John Doe";
    foaf:age "30";
    foaf:knows ex:Person2.

ex:Person2 rdf:type foaf:Person;
    foaf:name "Jane Smith";
    foaf:age "25";
    foaf:knows ex:Person1.
```

Figure 2: RDF Data Example in Turtle

Resource Description Framework Schema (RDFS) is an extension of RDF that enables it to represent more complex data, including classes, domains, and ranges of properties. There is also Web Ontology Language (OWL), which can represent even richer objects if needed [6]. These frameworks do not limit data serialization; many formats exist, such as Turtle, JSON-LD, RDF/XML, and HDT. All these formats have their trade-offs; for instance, allowing human readability can be helpful for debugging, but they add a lot of overhead data that might occupy more space. Thus, the flexibility of choosing a specific format can be part of the requirements [7].

2.1.2 Linked Data

Now that machines can understand the data, they still need to navigate through it, and that is where Linked Data comes into play. Linked Data allows the user to connect different resources; it leverages the RDF data model representation of ontologies to connect them to different ones. In doing so, it makes the resources extensible as well as reachable. According to [8], to enable the global sharing of resources, Linked Data employs the following criteria:

- The Uniform Resource Identifier (URI) is the resource name in the RDF representation.

- It would be even better if those URIs were online accessible via HTTP (valid URLs).

- Include URIs to other resources to connect the graphs.

- Dereferenced URIs return meaningful data that is compliant with standards such as RDF.

An example of a project that applies the Linked Data Principles is the Linked Open Data project, which started the Semantic Web in January 2007 and grew exponentially, as shown in Figure 3. This project is a community with the primary purpose of moving open data, which is publicly available, free-to-use data, to linked open data, connecting data, and allowing interoperability among the different entities [9].
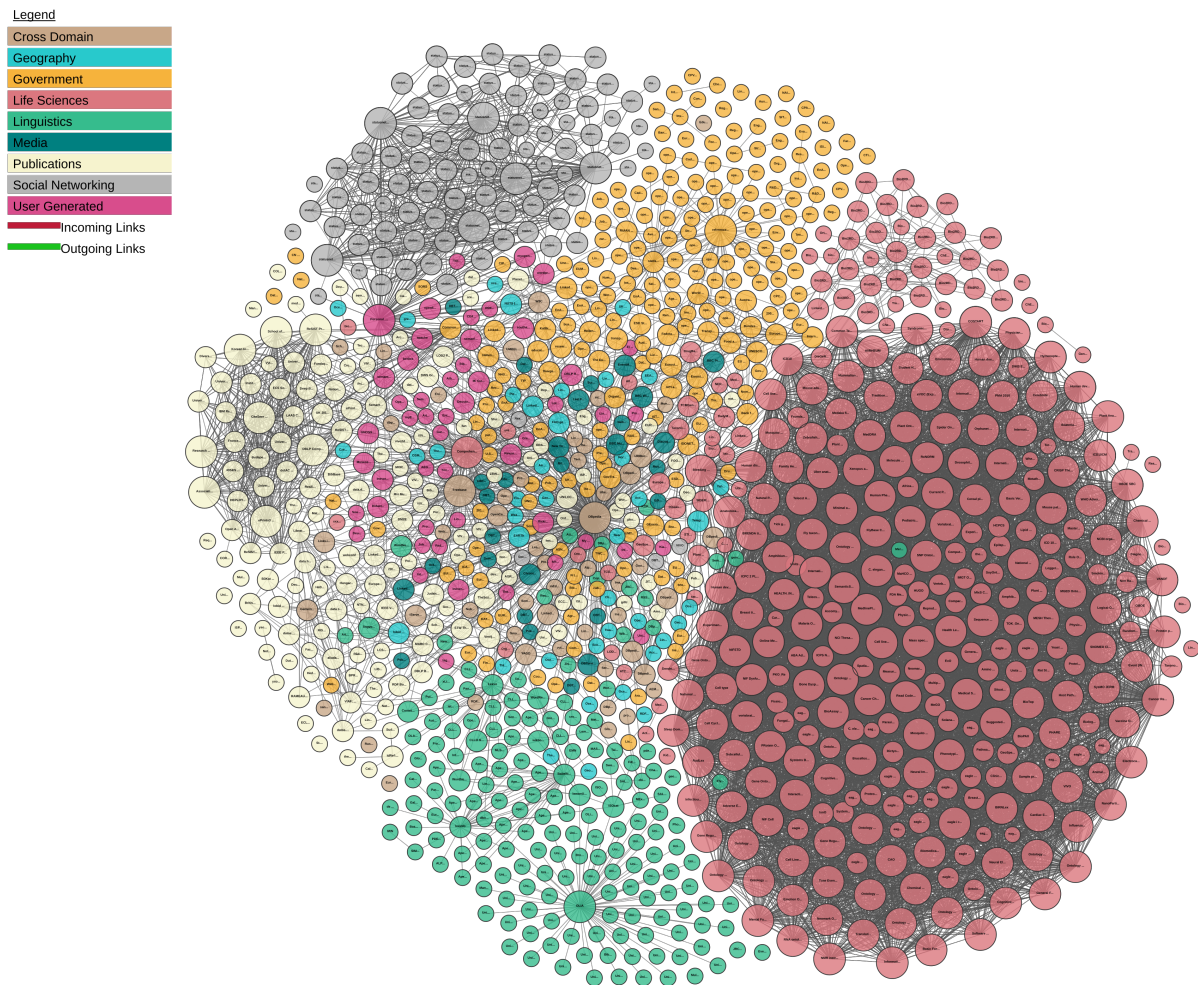

Figure 3: The Linked Open Data Cloud as of February 2017 [10].

2.1.3 SPARQL (SPARQL Protocol and RDF Query Language)

SPARQL (pronounced SPARKL) is a recursive acronym for SPARQL Protocol and RDF Query Language. It is a query language for RDF, similar to SQL for relational databases. The syntax is quite similar to SQL; it uses the SELECT statement to select a particular attribute from the data. It also uses the WHERE clause to narrow the data to a subset that meets a specific condition. The condition representation is in a triple form (subject-predicate-object). A variable can be bound to any part of the triple pattern and store its

query results. There are two ways to refer to a resource in a query:

- Using the Universal Resource Identifier (URI).
- Assigning the URI to a prefix and using the prefix instead.

The previously stated approaches can be beneficial because they shorten the query, especially when a long URI requires repeating multiple times, making it more human-readable [11]. Figure 4 shows this example of SPARQL using a Schema.org and DBpedia vocabulary. In this example, we query the RDF data shown in Figure 5 to retrieve books and their authors. The example also showcases the second approach, which involves assigning the URI to a prefix. Here, we assign the prefix for Schema.org to the "schema" prefix, thereby referencing the schema vocabulary. Figure 6 displays the query's output.

```
PREFIX schema: <http://schema.org/>

SELECT ?book ?authorName
WHERE {

  ?book a schema:Book ;
        schema:author ?author .
  ?author a schema:Person ;
          schema:name ?authorName .
}
```

Figure 4: Simple SPARQL query to retrieve books and their authors.

```
@prefix schema: <http://schema.org/> .

<https://dbpedia.org/resource/The_Fellowship_of_the_Ring> a schema:Book
↪  ;
    schema:author <https://dbpedia.org/resource/J._R._R._Tolkien> .

<https://dbpedia.org/resource/Harry_Potter_and_the_Philosopher%27s_Stone>
↪  a schema:Book ;
    schema:author <https://dbpedia.org/resource/J._K._Rowling> .

<https://dbpedia.org/resource/J._R._R._Tolkien> a schema:Person ;
    schema:name "J.R.R. Tolkien" .

<https://dbpedia.org/resource/J._K._Rowling> a schema:Person ;
    schema:name "J.K. Rowling" .
```

Figure 5: Simple Turtle of books and their authors.

Figure 6: The output of the query shown in Figure 4 for the data shown Figure 5.

2.2 Spatial Data Infrastructure (SDI)

Spatial Data Infrastructure (SDI) is an infrastructure that encapsulates all of the technologies, standards, frameworks, and workforce required to obtain, analyze, maintain, and share spatial data. The emergency of SDI stems from the vast amount of geospatial data decentralized among multiple government agencies and websites. This decentralization led to significant duplication of geospatial data collection, hence a lot of wasted resources (money, time, and effort). SDI exists at different geographical levels. It could be international, such as the Global Earth Observation System of Systems. It could be continent-wide, such as Infrastructure for Spatial Information in the European Community (INSPIRE). It could also be national, such as Data.gov, a data provider for the USA open dataset. SDI's main components include the portal, standards, framework, human resources, and data [12].

We will only concentrate on the standards component relevant to the thesis topic. SDI's standards allow a consistent, systematic way to serialize, exchange, and understand spatial data. They are necessary to allow interoperability across multiple organizations and platforms. The key is to choose widely accepted standards that meet the application's needs and can be used and shared by others without too much complexity. For spatial data in particular, the governing bodies responsible for such standards are mainly the Open GIS Consortium (OGC) and the International Standards Organization (ISO) [13].

Some of the data formats utilized to serialize and exchange spatial data, specifically 3D urban spatial data, that are certified by the OGC include:

- CityGML has been an OGC standard since March 2021 [14].

- CityJSON has been an OGC community standard[1], since November 2023 [15].

---

[1]A Community Standard is an official OGC standard developed and maintained outside of OGC. The originator of the standard brings to OGC a "snapshot" of their work, which is then endorsed by OGC membership as a stable, widely implemented standard that becomes part of the OGC Standards.

2.3   Background Summary

In this chapter, we discussed the necessary background information to follow the rest of the chapters. This chapter included a small introduction to the Semantic Web and its predecessors, followed by an explanation of the critical components that allow the Semantic Web to function correctly, such as ontologies, Linked Data, and SPARQL. We discussed ontology and its purpose in the Semantic Web. We discussed the principles that constitute Linked Data. We gave an example of SPARQL and how it looks to query data annotated with an ontology or vocabulary. Finally, we introduced SDI, its purpose, and how it links to the Semantic Web. The next chapter will discuss the standards used for 3D city modeling. We will look closer at CityGML, CityJSON, and JSON-LD and the reasons to convert CityJSON to JSON-LD.

3 Literature Review

This chapter provides more insight into CityGML, CityJSON, and JSON-LD, giving a glimpse of each's motivation and structure. The chapter also discusses work related to the thesis's topic.

3.1 CityGML

Before we explore CityJSON, it is crucial to grasp the significance of its predecessor, CityGML. Understanding these formats and their differences from existing ones is vital to appreciating their role in 3D urban modeling. This knowledge also highlights why CityJSON is a preferred choice over CityGML.

3.1.1 Overview

CityGML is a spatial data model and format specifically designed for modeling 3D urban objects. It is an OGC standard, as we mentioned before. It uses GML (Geography Markup Language), an OGC standard, with Extensible Markup Language (XML) to define its elements and structure. Any CityGML document is both a GML and an XML document, and GML is one of the most popular formats for storing geographical information. In contrast, XML is considered one of the most widely used formats for transferring data over the Web and storing data in both human and machine-readable formats [16].

3.1.2 Motivation

CityGML [16] emerged when the generation of 3D city data surged, primarily for visualization. Using such data for visualization only led to a lot of semantically rich information needing appropriate utilization. With that in mind, CityGML explicitly defines the relationships between each city object. A city object is anything found in a city, not just buildings.

3.1.3 The CityGML Data Model

The CityGML data model, according to [16], contains several sub-models, including the following:

- **Appearance Model**: This model handles the physical properties of objects, which can be visible (such as color and texture) or non-visible (such as infrared radiation and noise pollution).

- **Thematic Model**: This model represents city objects as classes based on widely found urban features such as vegetation, water bodies, bridges, and city furniture.

- **Spatial Model**: This model is responsible for geometrical and topological representation of city objects.

3.1.4 CityGML Levels of Detail (LoD)

As illustrated in Figure 7, CityGML supports multiple representations of a city object to cater to different purposes. It provides five levels of detail (LoDs), ranging from the most basic 2D representation to the most complex 3D representation. It allows for scalable

and flexible modeling, where a city object can be depicted with varying degrees of detail based on the application's specific requirements and serve different functionalities [16]. As described in [16] CityGML LoDs:

- LOD0: a basic 2D representation of an object.

- LOD1: a basic 3D representation of an object.

- LOD2: a more advanced 3D representation of an object; it includes roof shapes and minimal architectural detail.

- LOD3: a highly detailed 3D representation of an object that includes windows, doors, and other architectural features.

- LOD4: a highly detailed 3D representation of an object, the highest of which includes the interior design of the object.
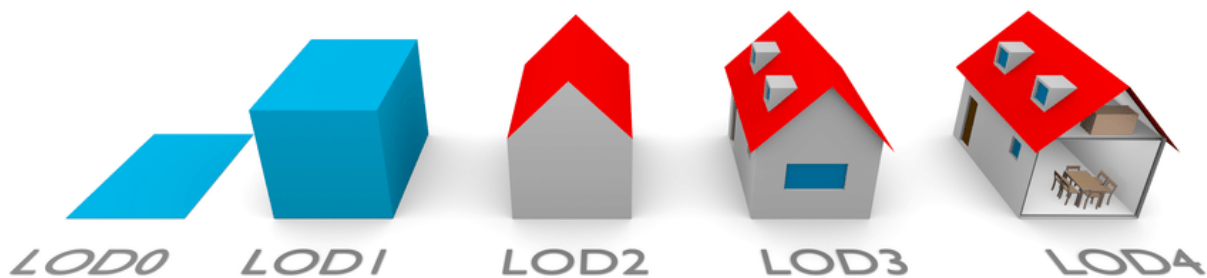

Figure 7: The five LODs of CityGML 2.0 as illustrated by F. Biljecki et al. [17]

3.2 CityJSON

Section 3.1.1 discussed how CityGML is a spatial data model and format. CityJSON, on the other hand, is another format that represents the CityGML data model. It uses JSON (JavaScript Object Notation) to encode the CityGML data model.

3.2.1 Overview

CityJSON is a JSON encoding of most of the CityGML data model to represent 3D urban data. CityJSON leverages the simple and human-readable syntax of JSON to represent such data. Using JSON as the encoding format, the model can be represented in a lightweight, developer-friendly way while maintaining the CityGML standards [18].

3.2.2 Motivation

Some of the flaws in the CityGML encoding that led to the development of CityJSON include [19]:

- CityGML is not developer-friendly due to the verbosity of the XML format.

- CityGML occupies significant storage, seven times the storage of CityJSON, again due to the XML language's verbosity and CityJSON compression capabilities.

- Due to XML's complexity, the same geometry can be represented in multiple ways, which could be problematic.

- There is no enforced standardization of CRS (Coordinate Reference System) in GML, meaning a valid CityGML file could theoretically have a building in one CRS and its doors in another. However, this is not considered best practice in CityGML.

3.2.3 Structure

CityJSON implements most of the CityGML modules discussed earlier in Section 3.1.3. However, some modules were left out to keep it simple and lightweight, such as the rarely used LoD4 to represent the interior of buildings. As the name suggests, CityJSON uses JSON, structured in single or nested objects as a key-value pair wrapped around curly braces, where the key is the property. The value is the value assigned to such a property, similar to that of a Python dictionary, and it uses an array list to store ordered data [19].

A CityJSON file **must** contain the following, as can be seen in Figure 8:

- Key "type" with the value "CityJSON".

- Key "version" with value a string of the form (X.Y).

- A "transform" object that contains a "scale" and "translate" object is required, as CityJSON stores the vertices as integers and applies a transformation function to retrieve their original value. This method maintains a small file.

- A "CityObjects" is an object of objects where each object is a unique city object that must contain a key "type" with a value that is one of the predefined first-level or second-level objects, as illustrated in Figure 9. If the city object is a second-level object, then a key "parent" with a value that corresponds to the identifier of the first-level object that is the parent of this object is required.

- An array of "vertices" that contains all the coordinates for all the objects in the city.

A CityJSON file **may** also contain:

- "metadata", which contains an object to describe the CRS of the entire object among other metadata attributes.

- "extensions", which contains an object representing the added extension.

- "appearance" which describes the texture or material of surfaces.

- "geometry-templates" refers to templates, allowing reusability for later.

- Any valid JSON object that is not part of the CityGML model, which CityJSON parsers will ignore.

A "CityObject" in a CityJSON file **may** contain the following:

- A "geometry" object, which must contain:
  - Key "type" with a value from one of the following: "MultiPoint", "MultiLineString", "MultiSurface", "CompositeSurface", "Solid", "MultiSolid", "CompositeSolid", or "GeometryInstance".
  - Key "lod" with a value of the form X(.Y), where X and Y can be any digits, and Y is optional.

13

– Key "boundaries" with a value of a nested array. The depth of the nesting depends on the "type" of geometry. The array contains the integer indices of the coordinates of the object in the vertex array.

- Key "attributes" and value any valid JSON object.

- Key "geographicalExtent" and value an array of six floats representing the min and max of each spatial axis.

- Key "children" and value an array of all the city objects in which this object is the parent.

```json
{
  "type": "CityJSON", //Must be CityJSON
  "version": "2.0", // CityJSON version 2.0 (latest)
  "transform": {
    "scale": [0.001, 0.001, 0.001], // Scale x, y, z coordinates by 0.001
    "translate": [2.5, 60.8, -1.804] // Shift x, y, z coordinates by 2.5, 60.8, -1.804 respectively
  },
  "CityObjects": {}, // Empty city objects
  "vertices": [] // Empty vertices array
}
```

Figure 8: Minimal valid CityJSON file

1st-level City Objects

Bridge
Building
CityFurniture
CityObjectGroup
GenericCityObject
LandUse
OtherConstruction
PlantCover
SolitaryVegetationObject
TINRelief
TransportationSquare
Railway
Road
Tunnel
WaterBody
Waterway
+Extension

2nd-level City Objects

BridgePart
BridgeInstallation
BridgeConstructiveElement
BridgeRoom
BridgeFurniture

BuildingPart
BuildingInstallation
BuildingConstructiveElement
BuildingFurniture
BuildingStorey
BuildingRoom
BuildingUnit

TunnelPart
TunnelInstallation
TunnelConstructiveElement
TunnelHollowSpace
TunnelFurniture

Figure 9: First and second-level CityJSON objects as illustrated by H. Ledoux et al. [19]

3.3 JSON-LD

3.3.1 Overview

JSON-LD is an RDF serialization format that uses JSON encoding to serialize data. Its development considered minimal changes needed to convert normal JSON to JSON-LD. The design of JSON-LD encourages developers to adopt a more Linked Data approach and enrich the semantics of the data exchanged.

3.3.2 Motivation

The exponential increase in the amount of data generated, as data nowadays is generated by humans and machines (for example, sensors and servers), led to the need for JSON-LD. One way to connect such data islands is by applying the Linked Data Principles [20]. Many existing data formats serialize data and apply such principles as Turtle, RDF, RDFa, RDF/XML, N-TRIPLES, N-QUADS, and TriG [21]. The advantage that separates JSON-LD from the previous formats is that it does not require many changes to the existing data serialized in JSON. More importantly, it eliminates the fear of using the Semantic Web, known as "Semaphobia" [20], by not requiring the developer to understand RDF and Linked Data. This reassurance allows developers to confidently use JSON-LD and its benefits, including integration with RDF tools such as SPARQL endpoint, reasoners, and triple stores.

3.3.3 Structure

As with all RDF formats, the JSON-LD structure is a directed label graph where each node connects via an edge. A node can be a primitive data type such as a string, number, date, or time. Alternatively, a node may be an Internationalized Resource Identifier (IRI) referencing an entity. An IRI is a more generalized form of a URI mentioned earlier in Chapter 2, which includes non-ASCII characters [22]. JSON-LD serializes the RDF model, classifying a node as a subject when an edge originates from it and an object when an edge terminates at it, where the edge represents the predicate. Although this violates the Linked Data Principle, JSON-LD permits non-dereferenceable IRIs for local referencing within the same document [20].

3.3.4 From JSON to JSON-LD

Extending a JSON document into a JSON-LD mainly requires some annotation on top of the already known JSON syntax; all other JSON data types are still valid [20]. As seen in Figure 11, there are only three different types of new keywords from the regular JSON syntax. We will discuss the essential keywords widely used in a JSON-LD context to convert JSON to JSON-LD, according to [23]:

- `@context`: defines the terms used later in the document. The terms are short-hand notations for IRIs, which help keep the document compact.

- `@type`: defines the type of the node using an IRI.

- `@id`: define the node identifier using an IRI or as a blank node.

- `@list`: defines ordered set of data.

By extending the human-readable CityJSON format to the machine-readable JSON-LD format, we leverage the advantages of both technologies. Those advantages allow us to improve 3D city modeling and spatial data. Such advantages may include:

- **Semantic Interoperability:** This would allow machines to exchange data without ambiguity or human input.

- **Improved Data Integration:** JSON-LD makes it simple to integrate data with systems that already use JSON and can link data from various sources.

- **Improved Data Sharing:** It allows the application of the **Linked Data Principles** and uses the standard JSON format, making sharing simple.

- **Enhanced Querying and Reasoning Capabilities:** JSON-LD's capabilities as an RDF serialization enable complex and advanced queries, inferences, and reasoning over the data.

- **Extensibility:** By converting the data to JSON-LD, adding new requirements is simple and can be done without significantly changing the underlying structure.

To illustrate the differences between JSON and JSON-LD, we will start with a simple JSON document representing a person, then add semantics to create an RDF representation with explicit semantics serialized as JSON-LD. Figure 10 shows a JSON that contains the following keys and values:

- key "name" and value "Alice".

- key "jobTitle" and value "Professor".

- key "telephone" and value "(425) 123-4567)".

- key "know" and value "http://www.bob.com".

However, the meanings of name, job title, and the other keys are vague and require more explicitness. For example, what does the key "name" represent? Is it the person's first, last, or family name? To add such explicitness, this is the role of a vocabulary. Let us assume we want Schema.org to represent people, their names, and their other attributes. *Schema.org* is just one of many vocabularies used to represent the concept of people, and one of the representations could be the one seen in Figure 11.

```
{

  "name": "Alice",
  "jobTitle": "Professor",
  "telephone": "(425) 123-4567",
  "knows": "http://www.bob.com"

}
```

Figure 10: Simple JSON example describing a person.
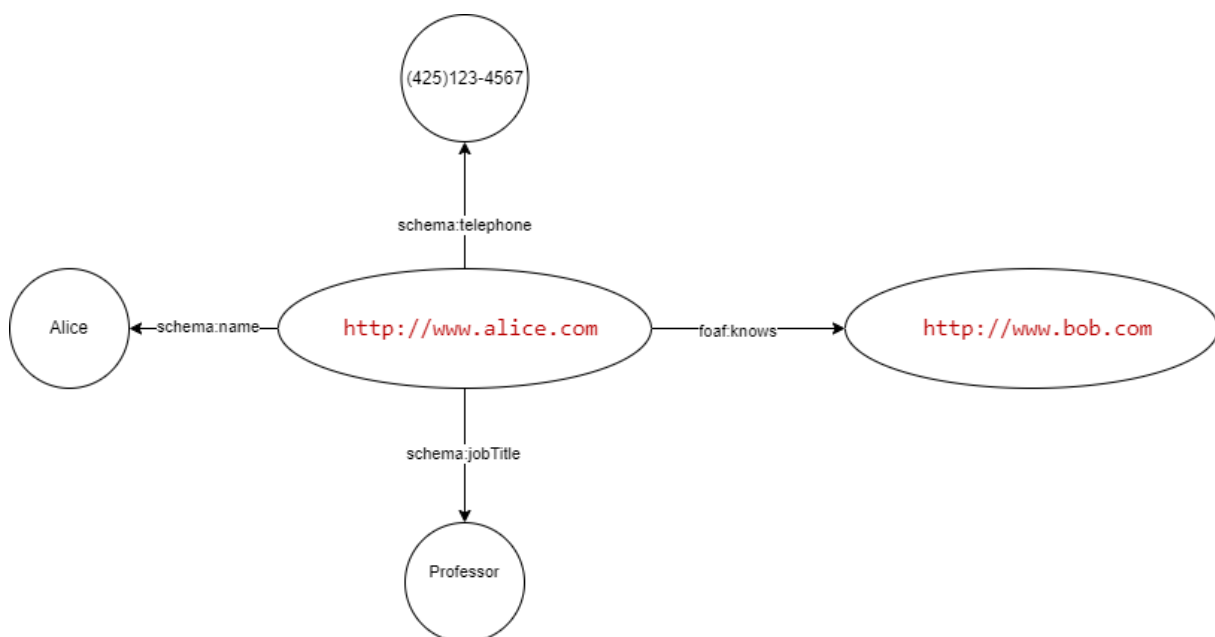
```
{
  "@context": {
    "schema": "http://schema.org/",
    "foaf": "http://xmlns.com/foaf/0.1/",
    "name": "schema:name",
    "jobTitle": "schema:jobTitle",
    "telephone": "schema:telephone",
    "knows": {
      "@id": "foaf:knows",
      "@type": "@id"
    }
  },

  "@type": "schema:Person",
  "@id": "http://www.alice.com",
  "name": "Alice",
  "jobTitle": "Professor",
  "telephone": "(425) 123-4567",
  "knows": "http://www.bob.com"
}
```

Figure 11: A JSON-LD representation of Figure 10.

Figure 12: RDF triple representation of Figure 11.

3.4 Review of Related Work

At the time of writing this thesis, there has yet to be an attempt to convert CityJSON into JSON-LD. The only related effort is a request on the W3C GitHub repository suggesting that CityJSON should rely on JSON-LD, but there have yet to be any advancements on this request. However, there have been attempts to convert CityJSON's predecessor, CityGML, to RDF, as evidenced by this paper [24]. The result is a comprehensive CityGML OWL ontology and the conversion of CityGML 2.0 to RDF, as shown in Figure 13.
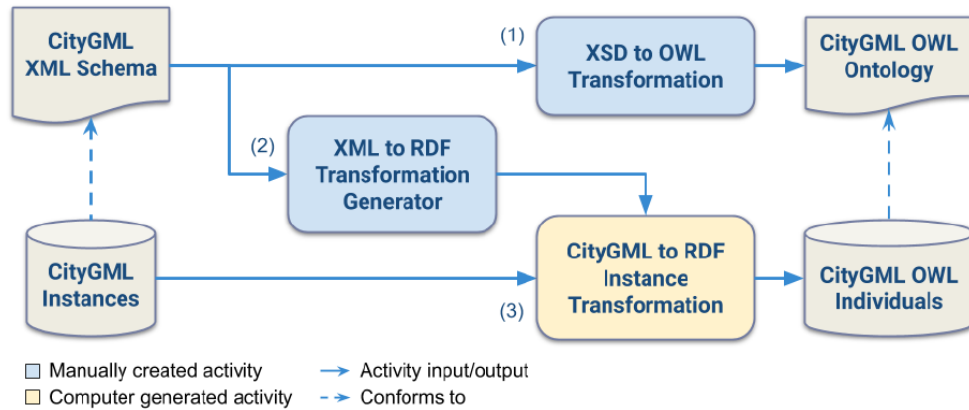
Figure 13: Overview of the pipeline for CityGML conversion according to [24].

The paper [25] resulted in another CityGML ontology in OWL and ontologies for IndoorGML and SensorThings API, as shown in Figure 14.
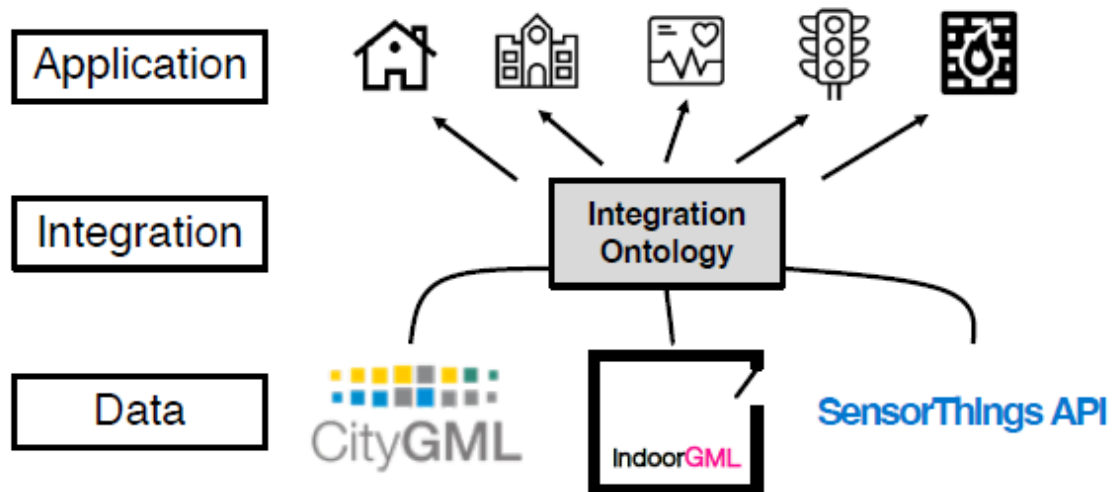
Figure 14: Overall framework of [25] solution.

The above papers have influenced this paper's approach to converting CityJSON to JSON-LD. Some of the methodologies for ontology engineering were adopted, as well as the conversion tool's approach.

3.5 Literature Review Summary

This chapter discussed relevant literature reviews, including the overview, motivation, and structure of CityGML, CityJSON, and JSON-LD, respectively. We also mentioned work related to the thesis and its impact on this thesis's methodology. In the upcoming chapter, we will discuss the approach taken to convert CityJSON to a JSON-LD RDF format.

4   Approach

In this chapter, we will discuss the practical part of this thesis by describing in detail the approach used to convert CityJSON to JSON-LD, as seen in Figure 15. The approach combines different technologies to help convert a CityJSON file into a semantically rich JSON-LD file. The conversion process incorporated the following technologies and tools: cjio tool for validating the CityJSON file and ensuring it complies with the CityJSON's schema and ISO 19107 geometric primitives for geometry representations. Protégé , a free, open-source ontology editor, was used to build the vocabulary used as a context in the JSON-LD output of our tool. Shapes Constraint Language (SHACL) is a W3C recommendation defining constraints the data needs to meet. Combined with the earlier vocabulary, they serve as the validation schema with which the CityJSON-LD output should comply. The output of this thesis is a conversion tool called "cj2jld". It is a Python-based CLI tool that takes a valid CityJSON file as input and converts it into a JSON-LD file; however, the tool does not support some optional CityJSON features such as appearances, geometry templates, extensions, and geometry semantics.
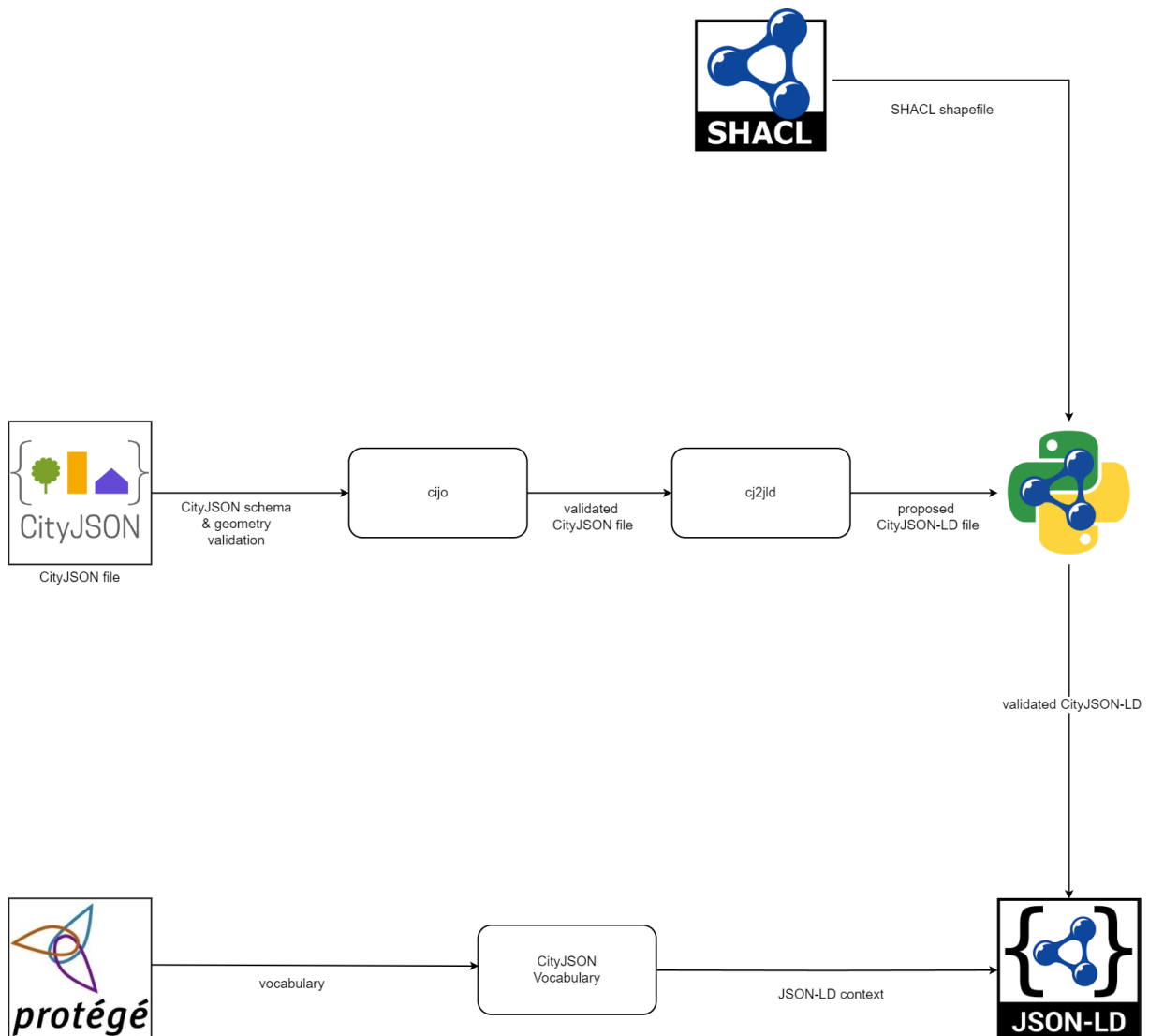
Figure 15: CityJSON to JSON-LD Overview

4.1 Vocabulary Engineering

Firstly, the distinction between a vocabulary and an ontology is in place, even though we often use them interchangeably. There is a subtle distinction between the two. A vocabulary utilizes RDFS and some "lightweight" OWL (e.g., inverseOf); an ontology, on the other hand, uses OWL with all its capabilities. However, it is crucial to understand that the tool solely showcases the potential benefits of converting CityJSON to JSON-LD, particularly the ability to leverage GeoSPARQL queries and their power on CityJSON data. Moreover, to create a more explicit representation of CityJSON that different technologies can use without needing a complicated conversion or a unique tool.

Using a vocabulary is well suited as it can adequately capture the essential relations between classes without needing more complex reasoning. Additionally, managing cardinality and integrity constraints using OWL can be cumbersome due to the open-world assumption. This assumption states that a missing piece of information in the knowledge base should be considered unknown, unlike in a relational database, which considers missing data false; this has significant implications for data interpretation and querying.

We used a top-down approach to create the vocabulary, map the general classes, and define their data properties from the CityJSON schema on their website before moving on to the more specific ones. An example would be mapping the CityJSON object to an RDF class, followed by the transform object, the city object, the vertices object, and finally, the metadata object, all of which have a relationship with the CityJSON object. Once we finished creating the classes, we began mapping their relations. This process started with the hierarchical definition of the sub-super classes, like the relationship between Cityobject (super-class) and FirstLevelCityObject (sub-class), and then proceeded to the mapping of object properties, such as the hasVertices relationship between Cityobject and Vertices classes.

The output shown in Figure 16 is a CityJSON vocabulary, which assumes that the vocabulary can be found online on this link "https://www.cityjson.org/ont/cityjson.ttl", which is not the case as the vocabulary is experimental for this thesis and is not available on the official website of CityJSON, it also assumes that the vocabulary is in *Turtle* format. We will continue to see the previous two assumptions throughout the thesis for illustration purposes. The resulting vocabulary contains the set of CityJSON terms that will provide the context for the generated JSON-LD. It encompasses all the necessary features of CityJSON and includes a 2D well-known text (WKT) representation of the geometry within CityJSON to enable the use of GeoSPARQL. It also makes the CityJSON data more explicit by containing the relations between objects; it also adds more details regarding vertices coordinates used in the geometry object; it makes it more explicit with the actual coordinates being shown rather than the coordinate index in the vertices array.

The vocabulary contains metadata information to be used by machines in order to make well-structured suggestions; an example would be the *seeAlso* property of RDFS that would enable a machine to suggest related topics to the user, which is similar or deemed to be similar by the authors of the vocabulary to the object currently viewed. Another example is assigning the language to the label property, which is helpful for the machine to suggest the relevant language to the user.

```
################################################################
#     Classes
################################################################
...
###   https://www.cityjson.org/ont/cityjson.ttl#Bridge
:Bridge rdf:type owl:Class ;
        rdfs:subClassOf :FirstLevelCityObject ;
        rdfs:comment "The Bridge module provides the representation of thematic and
        ↪   spatial aspects of bridges. Bridges are movable or unmovable structures
        ↪   that span intervening natural or built elements. In this way, bridges
        ↪   allow the passage of pedestrians, animals, vehicles, and service(s) above
        ↪   obstacles or between two points at a height above ground."@en ;
        rdfs:isDefinedBy :Bridge ;
        rdfs:label "Bridge"@en ;
        rdfs:seeAlso <https://docs.ogc.org/is/20-010/20-010.html#toc44> ,
                      <https://www.cityjson.org/specs/2.0.0/#bridge> .
###   https://www.cityjson.org/ont/cityjson.ttl#BridgeConstructiveElement
:BridgeConstructiveElement rdf:type owl:Class ;
                           rdfs:subClassOf :BridgeSecondLevelObjects ;
                           rdfs:isDefinedBy :BridgeConstructiveElement ;
                           rdfs:label "Bridge Constructive Element"@en .

###   https://www.cityjson.org/ont/cityjson.ttl#BridgeFurniture
:BridgeFurniture rdf:type owl:Class ;
                 rdfs:subClassOf :BridgeSecondLevelObjects ;
                 rdfs:isDefinedBy :BridgeFurniture ;
                 rdfs:label "Bridge Furniture"@en .

###   https://www.cityjson.org/ont/cityjson.ttl#BridgePart
:BridgePart rdf:type owl:Class ;
            rdfs:subClassOf :BridgeSecondLevelObjects ;
            rdfs:isDefinedBy :BridgePart ;
            rdfs:label "Bridge Part"@en .

###   https://www.cityjson.org/ont/cityjson.ttl#BridgeRoom
:BridgeRoom rdf:type owl:Class ;
            rdfs:subClassOf :BridgeSecondLevelObjects ;
            rdfs:isDefinedBy :BridgeRoom ;
            rdfs:label "Bridge Room"@en .

###  https://www.cityjson.org/ont/cityjson.ttl#BridgeSecondLevelObjects
:BridgeSecondLevelObjects rdf:type owl:Class ;
                          rdfs:subClassOf :SecondLevelCityObject ;
                          rdfs:isDefinedBy :BridgeSecondLevelObjects ;
                          rdfs:label "Bridge Second-Level Objects"@en .

###   https://www.cityjson.org/ont/cityjson.ttl#BrigeInstallation
:BrigeInstallation rdf:type owl:Class ;
                   rdfs:subClassOf :BridgeSecondLevelObjects ;
                   rdfs:isDefinedBy <https://www.cityjson.org/ont/cityjson.ttl#> ;
                   rdfs:label "Bridge Installation"@en .
 ...
```

Figure 16: CityJSON's Vocabulary[2] in Turtle

4.2 SHACL

SHACL, a W3C recommendation designed to validate RDF [26], was required in addition to the vocabulary to create a SHACL shape file, which contains the shape of how the JSON-LD output should look. It will be used as a validation schema for the tool to ensure that the output matches the vocabulary generated earlier, as shown in Figure 17. SHACL provided functionalities that allowed to keep the vocabulary simple and yet defined the constraints without the need to use OWL and to deal with the open-world assumptions mentioned earlier in Section 4.1.

The SHACL file was designed based on the CityJSON specification in their documentation and the JSON schema supplied on their website, as shown in Figure 18. The schema supplied the following:

- The cardinality for each object.

- The type of each object.

- The enumeration values for an object, if it has any.

- The pattern for the value of an object, if it has any.

- The format of the value of an object, if it has any.

- The mandatory and optional objects.

SHACL Playground was used as the platform to develop the SHACL file used by the conversion tool. A top-down approach was applied to generate the SHACL file, starting with each separate object and creating its constraints, then referring to those shapes inside other objects. We continued the process until we met all constraints and created all object shapes.

4.3 Data Collection

We acquired data from the CityJSON's official website; the website contained simple examples of different geometric shapes, such as cube and tetrahedron shapes, as well as data from real cities such as New York, Deen Haag, Ingolstadt, Montréal, Rotterdam, Vienna, and Zürich. Nevertheless, we obtained another dataset from this repository on GitHub. The dataset represents the city of Helsinki, located in Finland. It was converted from CityGML to CityJSON using citygml-tools. After removing the appearance object and manually adding the CRS, it was tested by uploading to CityJSON Ninja and CityJSON Up3date for visualization. However, both tools validate their input before usage, so a successful upload means a valid CityJSON file.

4.4 Data Preprocessing

Before the collected data was considered valid testing data for the conversion tool, it first went through a preprocessing step by jq, a command-line JSON processor, to remove all the unsupported objects. This step was needed to avoid discarding valid data, and it was also necessary because the publicly available CityJSON data are scarce and hard to find, so it was necessary to utilize as much of the data available as possible.

```
###############################################################
#    FirstSecondLevelCityObject Shape
###############################################################

cj:FirstLevelCityObjectShape
    a sh:NodeShape ;
    sh:targetClass cj:FirstLevelCityObject ;
    sh:closed true ;
    sh:property [
        sh:path rdf:type ;
        sh:hasValue cj:FirstLevelCityObject
    ] ;
    sh:property [
        sh:path cj:type ;
        sh:in (
                "Bridge"
                "Building"
                "CityFurniture"
                "CityObjectGroup"
                "GenericCityObject"
                "LandUse"
                "Metadata"
                "OtherConstruction"
                "PlantCover"
                "Railway"
                "Road"
                "SolitaryVegetationObject"
                "TINRelief"
                "TransportationSquare"
                "Tunnel"
                "Waterbody"
                "Waterway"
            ) ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path cj:hasAttribute ;
        sh:minCount 0 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path cj:hasGeographicalExtent ;
        sh:node cj:GeographicalExtent ;
        sh:minCount 0 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path cj:hasChildren ;
        sh:node cj:SecondLevelCityObject ;
        sh:minCount 0 ;
    ] ;

    sh:property [
    sh:path cj:hasGeometry ;
    sh:node cj:GeometryShape ;
    sh:minCount 0 ;
    ] .
```

Figure 17: SHACL for CityJSON's FirstLevelCityObject in Turtle

```json
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "$id": "https://www.cityjson.org/schemas/2.0.1/metadata.schema.json",
    "title": "CityJSON metadata schema v2.0.1",
    "definitions": {
        "contactDetails": {
            "type": "object",
            "properties": {
                "contactName": { "type": "string" },
                "phone": { "type": "string" },
                "address": { "type": "object" },
                "emailAddress": { "type": "string", "format": "email" },
                "contactType": { "type": "string", "enum": ["individual", "organization"] },
                "role": {
                    "type": "string",
                    "description": "Role of the contact as per ISO 19115 codelist",
                    "enum": [
                        "resourceProvider",
                        "custodian",
                        "owner",
                        "user",
                        "distributor",
                        "originator",
                        "pointOfContact",
                        "principalInvestigator",
                        "processor",
                        "publisher",
                        "author",
                        "sponsor",
                        "co-author",
                        "collaborator",
                        "editor",
                        "mediator",
                        "rightsHolder",
                        "contributor",
                        "funder",
                        "stakeholder"
                    ]
                },
                "organization": { "type": "string" },
                "website": { "type": "string", "format": "uri", "pattern": "^(https?)://" }
            },
            "required": ["contactName", "emailAddress"]
        }
    },
    "metadata": {
        "type": "object",
        "properties": {
            "identifier": { "type": "string" },
            "pointOfContact": { "$ref": "#/definitions/contactDetails" },
            "referenceDate": { "type": "string", "format": "date" },
            "title": { "type": "string" },
            "geographicalExtent": { "type": "array", "items": { "type": "number" }, "minItems": 6,
            ↪   "maxItems": 6 },
            "referenceSystem": { "type": "string", "pattern": "^(http|https)://www.opengis.net/def/crs/"
            ↪   }
        }
    }
}
```

Figure 18: CityJSON metadata schema

4.5 Design of the Conversion Process

In this section, we will discuss the design of the conversion tool. The tool depends on the vocabulary and the SHACL shape file mentioned earlier in Section 4.1. The design process consists of several steps, as seen in Figure 19. Below, we outline the detailed steps involved in the conversion process.

4.5.1 Validation of the input file

Using the Python cjio package, the tool first determines whether the input file is a valid CityJSON. If the file is invalid, the tool displays an error message and terminates; otherwise, it advances to the next step.

4.5.2 Supported Feature Check

The tool checks for non-supported features in the file; if such features exist, the tool displays an appropriate error message and terminates.

4.5.3 Field Extraction and Object Instantiation

If the non-supported features are not in the supplied CityJSON input file, the tool extracts the required fields and supplies them as arguments to the class constructor corresponding to the extracted CityJSON object.

4.5.4 Conversion Process

After completing the previous step, the result would be a CityJSON class object populated with the data that the input CityJSON contained; the class includes a to_json method that, when invoked, returns a JSON-LD representation of the CityJSON object. A process we will elaborate on more thoroughly in Section 4.6.

4.5.5 SHACL Validation

If the user sets the `--enable-pyshacl` flag up, we use the resulting JSON-LD representation of the CityJSON as input to the PySHACL package. The PySHACL package returns a boolean that is true when the data conforms with the supplied SHACL file and false otherwise; it also returns a validation report containing the details of the violation, if there are any.

4.5.6 Output Formatting

If the user has set the `--formatted flag`, the system writes the conforming data in a formatted manner to the file; if not, it saves the data in its raw form. If the data does not conform, the console prints the violation report.

4.5.7 Design Summary

We designed the tool to ensure a user-friendly and straightforward conversion process that is flexible enough to handle various scenarios and provides clear feedback at each stage. The tool uses Python packages such as cjio and PySHACL for the optimal validation and

transformation of CityJSON data, giving the user the power and flexibility to work with CityJSON files and Linked Data technologies.
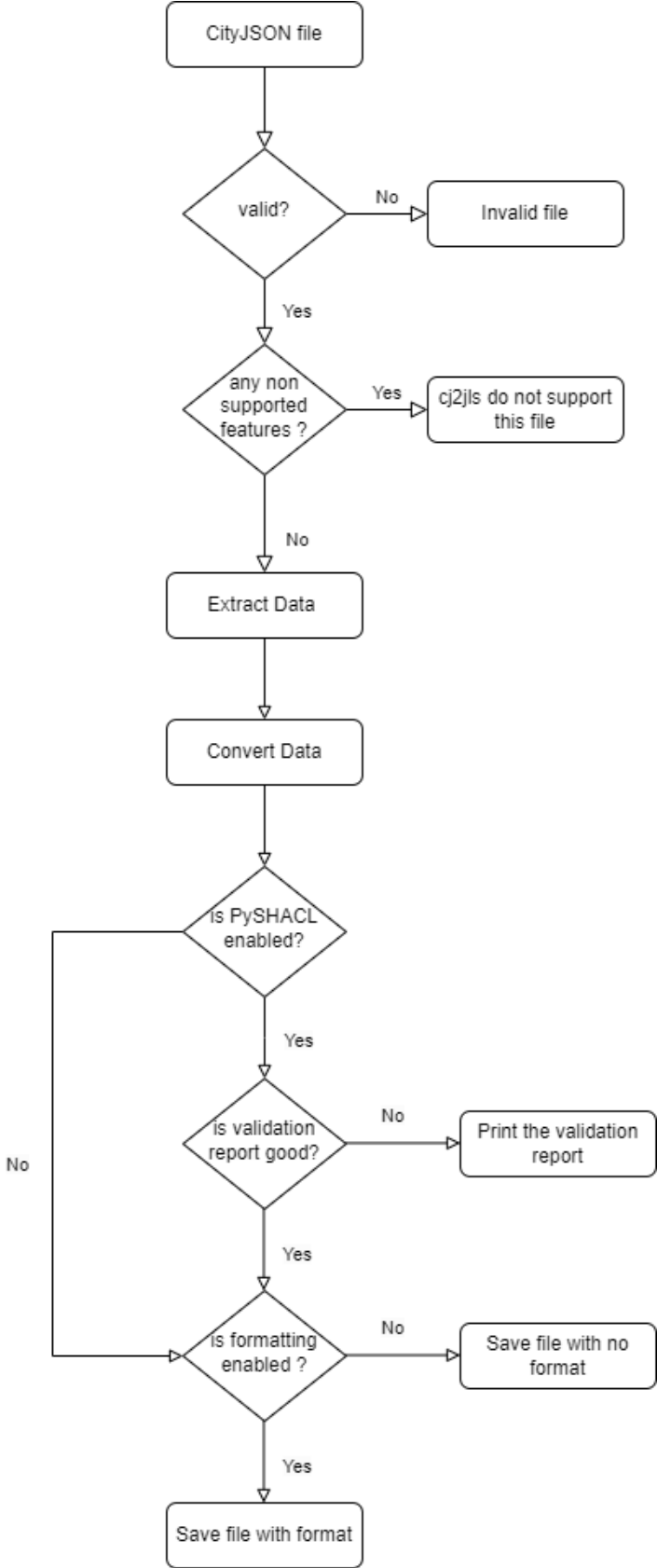
Figure 19: cj2jld Design Flowchart

4.6 Conversion to JSON-LD

We adopted an object-oriented approach by creating a class for each CityJSON object. The approach is similar to the *Builder Pattern*, as our conversion tool creates different representations of a complex object. In this case, it is the CityJSON class; the sole purpose of each class was to convert its content into a valid JSON-LD representation that follows the previously designed vocabulary and SHACL file. The previous is the case for most objects, with an additional requirement for the geometry object on top of the conversion, which is constructing a WKT string, as seen in Figures 20 and 21. WKT is a lightweight format for encoding geometric vectors in machine-readable and human-readable simple text [27]. A mapping between CityJSON geometries and the geometries represented by WKT was needed, as indicated in Table 1.

CityJSON Geometry	WKT Geometry
MultiPoint	MULTIPOINT
MultiLineString	MULTILINESTRING
MultiSurface	MULTIPOLYGON
CompositeSurface	MULTIPOLYGON
Solid	MULTIPOLYGON
MultiSolid	MULTIPOLYGON
CompositeSolid	MULTIPOLYGON

Table 1: Table mapping CityJSON geometry to WKT

As mentioned earlier in Section 4.1, one of the main goals of this conversion is to allow the usage of GeoSPARQL. For that, we chose the WKT format as GeoSPARQL supports it, and it is also one of the widely used formats to represent geometry in RDF [28], but it supports only 2D geometries. We needed to apply projection to the 3D geometry to convert it to 2D. Our initial approach was as follows:

- Remove the Z-Coordinate from the 3D array and convert it into a 2D array.

- Apply a convex hull function to capture the simplest multi-polygon that can encompass all the polygons inside of it.

However, we found that this approach needed to be revised. The convex hull approach resulted in an inaccurate representation of the city objects' boundaries. This inaccuracy could be because of how each city object is encoded individually without complex objects within each other, and by applying the convex hull, we lose information rather than maintaining the geometry without losing much information; therefore, we realized that the projection alone was a sufficient approach, as indicated by Figures 22b and 22a, which compare the visualization of two solid buildings from the Helsinki dataset in WKT.

```
{
    ...

    "geometry": [
        {
            "type": "MultiPoint",
            "lod": "1",
            "boundaries": [ 1, 2, 3 ]
        }
    ]

        ...
}
```

Figure 20: An example of a MultiPoint Geometry object in CityJSON

```
{
    ...

    "cj:hasGeometry": [
        {
            "@type": "cj:Geometry",
            "cj:type": "MultiPoint",
            "cj:lod": "1",
            "geosparql:asWKT": {"@value": "MULTIPOINT (1 0, 1 1, 0 1)", "@type": "geosparql:wktLiteral"},
            "cj:hasBoundingBox": {
                "@type": "cj:MultiPoint",
                "cj:hasPoint": [
                    {
                        "@type": "cj:Point",
                        "cj:boundaryX": { "@value": 1.0, "@type": "xsd:float" },
                        "cj:boundaryY": { "@value": 0.0, "@type": "xsd:float" },
                        "cj:boundaryZ": { "@value": 1.0, "@type": "xsd:float" }
                    },
                    {
                        "@type": "cj:Point",
                        "cj:boundaryX": { "@value": 1.0, "@type": "xsd:float" },
                        "cj:boundaryY": { "@value": 1.0, "@type": "xsd:float" },
                        "cj:boundaryZ": { "@value": 1.0, "@type": "xsd:float" }
                    },
                    {
                        "@type": "cj:Point",
                        "cj:boundaryX": { "@value": 0.0, "@type": "xsd:float" },
                        "cj:boundaryY": { "@value": 1.0, "@type": "xsd:float" },
                        "cj:boundaryZ": { "@value": 1.0, "@type": "xsd:float" }
                    }
                ]
            }
        }
    ]

    ...
}
```

Figure 21: Example 20 converted to JSON-LD, in accordance with our Vocabulary

(a) Without convex hull applied

(b) With convex hull applied

Figure 22: A Comparison between WKT without and with convex hull applied

4.7 Approach Summary

This chapter discussed the approach to implementing the vocabulary, SHACL file, and the CityJSON to JSON-LD conversion tool. We talked about the overview design, architecture, and technologies used. We documented how the data was collected and prepossessed before use and explained in detail the conversion of CityJSON geometry to a WKT string. In the next chapter, we will take a deep dive into the technical aspect of the tool in the implementation chapter.

5   Implementation

In this chapter, we will discuss the implementation of the CLI conversion tool, starting with the creation of the vocabulary with the help of the Protégé ontology editor, followed by the creation and utilization of the Unified Modeling Language (UML), depicted in Figure 24. The UML is a class diagram displaying the used Python classes and the relationships they have with each other. Next, we delve deeper into the conversion tool's arguments and the features it can use when it receives them. We also delve into validating and testing the conversion tool's output, emphasizing the significance of PySHACL in this process and its preference over other SHACL validation tools. Finally, we discuss the role of the Shapely Python package in generating and validating the WKT representation of the CityJSON city object's geometry.

To illustrate the tool's conversion of a CityJSON file into a JSON-LD using the vocabulary in Section 4.1. We will discuss an example; in the example, we will assume that the input file supplied to the tool contains a `Multipoint` geometry that requires conversion to JSON-LD. Figure 23 shows the `Geometry` class. The class will work as follows:

- First, the constructor is called to initialize the class instance variables with the provided values for type, load, boundaries, vertices, scale, and translate.

- The constructor also calls a class method called to_wkt, which takes the following input: boundaries, vertices, scale, and translate. Based on the type, it calls the method responsible for returning the corresponding WKT string using the Shapely Python package. In this case, it is the `point_to_wkt`.

- The `point_to_wkt` method takes boundaries, vertices, scale, and translate, and it calls the `get_real_vertex`; this function is responsible for retrieving the actual coordinate from the vertices array by using the boundaries which contain the actual location of the coordinates in the vertices array. After that, it applies the transformation function which corresponds to this equation:

$$vertex[i] * scale[i] + translate[i]$$

  where i is either 0, 1, or 2, corresponding to the location of the x,y, and z coordinates in the vertices array.

  The method returns both a version of the boundaries with actual coordinates and the WKT string.

- `to_wkt` uses the returned boundaries to be passed to the constructor for the appropriate class representing that type; in the example, it is the `MultiPoint` class.

- Finally, the class initialization step is complete, and the `to_json` method is ready to be called to provide the complete JSON-LD representation of the `Geometry` object.

```
class Geometry:
    """
    A class to represent a cityJSON geometric object with various types such as MultiPoint, MultiLineString, MultiSurface,
    CompositeSurface, Solid, MultiSolid, CompositeSolid.e
    Attributes:
        type (str): The type of the geometric object.
        lod (str): The level of detail of the geometric object.
        boundaries (List): The boundaries of the geometric object in CityJSON format.
        boundingBox (CjMultiPoint | CjMultiLineString | CjMultiCompositeSurface | CjSolid | CjMultiCompositeSolid): The bounding box of the
        ↪  the geometric object
    Arguments:
        type (str): The type of the geometric object.
        lod (str): The level of detail of the geometric object.
        boundaries (List): The boundaries of the geometric object.
        vertices: The vertices of the geometric object.
        scale: The scale factor for the vertices.
        translate: The translation vector for the vertices.
    """

    def __init__(self, type: str, lod: str, boundaries, vertices, scale, translate):
        self.type = type
        self.lod = lod
        self.boundaries = self.to_wkt(boundaries, vertices, scale, translate)

    def get_real_vertex(self, index, vertices, scale, translate):
        """
        Replace vertex index with actual vertex coordinates and apply the transform function.
        :param index: The index of the vertex.
        :param vertices: List of vertices.
        :param scale: The scale factors.
        :param translate: The translation factors.
        :return: Transformed vertex.
        """
        vertex = vertices[index]
        return (vertex[0] * scale[0] + translate[0], vertex[1] * scale[1] + translate[1], vertex[2] * scale[2] + translate[2])

    def point_to_wkt(self, points, vertices, scale, translate):
        """
        Convert (Multi)points to WKT format.
        :param points: List of points.
        :param vertices: List of vertices.
        :param scale: The scale factors.
        :param translate: The translation factors.
        :return: (Multi)Points in 2D WKT format and (Multi)Points in 3D coordinates in JSON-LD format.
        """
        points_as_vertices_3d = [self.get_real_vertex(point, vertices, scale, translate) for point in points]
        # Convert to 2D by ignoring the Z-coordinate
        points_as_vertices_2d = [(pt[0], pt[1]) for pt in points_as_vertices_3d]
        multi_point = MultiPoint(points_as_vertices_2d)
        return multi_point.wkt, points_as_vertices_3d

    ...

    def to_wkt(self, boundaries, vertices, scale, translate):
        """
        Convert the cityJSON geometry boundaries to WKT format.
        :param boundaries: The boundaries of the geometric object.
        :param vertices: The vertices of the geometric object.
        :param scale: The scale factors.
        :param translate: The translation factors.
        :return: Boundaries in WKT format.
        """
        if self.type == "MultiPoint":
            multi_point_wkt, multi_point_3d = self.point_to_wkt(boundaries, vertices, scale, translate)
            self.boundingBox = CjMultiPoint(multi_point_3d)
            return multi_point_wkt

        ...

    def to_json(self):
        """
        Convert the CityJSON Geometry object to a JSON-LD representation.
        :return: JSON-LD representation of the Geometry object.
        """
        data = {
            "@type": "cj:Geometry",
            "cj:type": self.type,
            "cj:lod": self.lod,
            "geosparql:asWKT": {
                "@value": self.boundaries,
                "@type": "geosparql:wktLiteral"
            },
            "cj:hasBoundingBox": self.boundingBox.to_json()
        }
        return data
```

Figure 23: Geometry Class[3]

Figure 24: Class diagram of the conversion tool design

5.1 Protégé

We used the Protégé tool to create the CityJSON vocabulary. The tool is one of the most widely used ontology editors [29], available as a desktop and web application. Such a tool was beneficial, as it contains numerous tools that facilitate an ontology's organized and straightforward construction. Such tools include the OntoGraf to visualize the ontology, as seen in Figure 25. Protégé also features multiple configurable reasoners for inferring axioms and rules and validating the ontology, as well as various plug-ins developed for visualization, validation, and other purposes. The tool allows the user to export the ontology in many formats, such as Turtle, RDF/XML, OWL/XML, OWL Functional Syntax, Manchester OWL Syntax, OBO Format, and JSON-LD. It also helped import other ontologies, like the GeoSPARQL ontology, which added a WKT representation to the CityJSON object geometry.

Additionally, it includes standard ontologies such as RDF, RDFS, OWL, and XML by default and allows adding external ontologies either by providing their IRIs or by uploading them. After using Protégé to create the basic vocabulary and exporting it in Turtle format because it was the most straightforward and human-readable format, the next step was to develop a SHACL shape file to set the constraints and provide a way to check the tool's output based on the developed vocabulary.
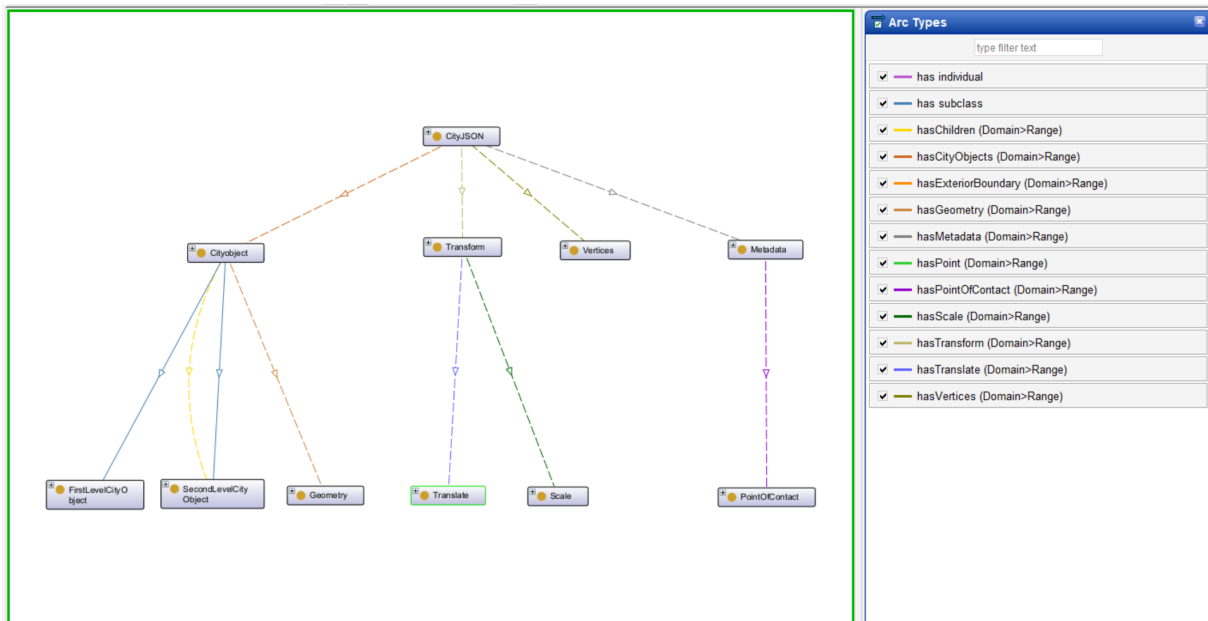
Figure 25: Protégé's OntoGraf visualization of CityJSON vocabulary

5.2 Conversion Tool

The CLI Python tool performs the following:

- Accept a valid CityJSON file without appearance, extensions, or semantics. Otherwise, it should throw a descriptive, not support, message.

- Generate a valid CityJSON-LD for a given CityJSON file.

- Allow the usage of the following argument:

  - **-i** or **--input-file**: Allow the user to provide the input CityJSON file path, relative or absolute; if relative, the file should be in the src/data folder (required).

  - **-o** or **--output-file**: Allow the user to provide the output JSON file path relative or absolute; if relative, the tool will place the file in the output folder inside the src/output folder (required).

  - **-b** or **--base-url**: Allow the user to provide the base URL (required).

  - **-id**: Allow the user to provide the ID of the supplied CityJSON file input (required if **-idp** argument not provided).

  - **-idp** or **--id-path**: Allow the user to provide the path to a JSON file containing a metadata object with an identifier key representing the ID of the supplied CityJSON file input (required if **-id** not supplied).

  - **-epyshacl** or **--enable-pyshacl**: Allow the user to enable PySHACL validation (Warning: could potentially take a significant amount of time with big files; not recommended by default, false).

  - **-f** or **--formatted**: Allow the user to enable formatting or beautifying the JSON-LD into a human-readable format; we do not recommend it, as it takes too much space by default; it is false.

  - **-h** or **--help**: This argument provides a manual for the tool, guides the user through its use, and provides documentation for the arguments mentioned earlier.

- Convert the CityJSON geometry in the file to a valid WKT format geometry supported by GeoSPARQL endpoints and replace the CityJSON geometry vertices indices array with the actual values of the vertices.

The tool takes into account all mandatory CityJSON features as well as most optional ones. The tool lacks support for the following features: appearance, geometry templates, extensions, and semantics of geometric primitives.

5.3 Validation and Testing Procedures

This section will discuss the validation and testing procedures used for the conversion tool. The validation procedure consists of two steps. The first step involves validating a CityJSON file as the tool's input. The second part involves validating the tool's output using PySHACL against the previously mentioned SHACL shape file.

5.3.1 cjio

The initial step entails validating the CityJSON file, the tool received as input. For this, we utilize the cjio tool, which is a Python CLI to process and manipulate CityJSON files; cjio can validate the CityJSON file supplied to it; it validates against the CityJSON schema and the validity of the geometry; it then returns a boolean on whether this is a valid file or not; in that case, we return an appropriate message in case the file is invalid, or we continue processing the file otherwise.

5.3.2 WKT

There was also a testing procedure for the WKT string generated in the geometry of a city object to ensure the string's validity before testing the WKT representation of the geometry. We confirmed the WKT string by using the Shapely Python package based on the well-known GEOS to generate the WKT string of the spatial object. Next, we conducted user testing to verify the correct correspondence between the generated WKT and the spatial object it described. The validation included copying the WKT string to a WKT visualization tool; the one used was the online WKT Map. The tool takes the WKT string along with the CRS used and gives a visualization of the object on a map. We then tested manually to check that the generated WKT string correctly reflects the actual spatial object on the map.

5.3.3 PySHACL

After the conversion tool converts CityJSON to JSON-LD, the user can apply an optional step: testing using PySHACL, a Python tool that validates any RDF file of any format against a supplied SHACL shape file; the result is a validation report that contains all the violations in the provided input file regarding the supplied SHACL shape file, as seen in Figure 26.

The choice of using PySHACL for validating the JSON-LD output of the tool stems from the following:

- Easily imported Python package, the same language the tool uses.

- Flexibility to change shapes maximum recursion depth.

- It does not stop after the first violation and provides a full report of all the violations [30].

```
Data does not conform to SHACL shapes. Validation errors:
Validation Report
Conforms: False
Results (1):
Constraint Violation in InConstraintComponent
↪  (http://www.w3.org/ns/shacl#InConstraintComponent):
        Severity: sh:Violation
        Source Shape:
                [
                sh:datatype xsd:string ;
                sh:in ( Literal("CityJSON") ) ;
                sh:maxCount Literal("1", datatype=xsd:integer) ;
                sh:message Literal("Only one type should exist and the
                ↪  value must be "CityJSON"") ;
                sh:minCount Literal("1", datatype=xsd:integer) ;
                sh:path cj:type
                ]
    Focus Node: ex:eaeceeaa-3f66-429a-b81d-bbc6140b8c1cclea
    Value Node: Literal("CityJSONE")
    Result Path: cj:type
    Message: Only one type should exist and the value must be "CityJSON"
```

Figure 26: SHACL validation report example

This step is optional because our vocabulary includes a list of vertices, which must be defined in a recursive method to validate each vertex in the list of vertices, ensuring that each vertex shape aligns with the SHACL shape that describes it. The problem is that recursion shapes in SHACL are not defined, and according to W3C, the SHACL processor implementation determines how to handle recursion however they see fit [31]. In the case of a large city with many vertices encoded in its CityJSON file, recursion could lead to exceeding the maximum recursion depth set by the SHACL tool. In PySHACL, the **--max-depth** argument allows the user to set the maximum depth; if **--enable-pyshacl** is present in the conversion tool arguments, the tool sets the **--max-depth** argument to 999 in order to avoid the need for more depth. We recommend using PySHACL only for small CityJSON files, as larger ones may take too long and fail.

Technically, enabling PySHACL is unnecessary, as the tool has undergone testing across multiple scenarios and corner cases, ensuring that the output does not violate the SHACL shape file. Since different SHACL processors might have different implementations, the test cases were also run on processors other than PySHACL, such as Apache Jena SHACL, and SHACL processors found online, like SHACL Playground. All of these gave the same results, but the selection of PySHACL was due to the reasons explained earlier in Section 5.3.3.

5.4 Implementation Summary

To summarize, in this chapter, we talked about the implementation of the conversion tool; we discussed the class diagram used as a blueprint to create the Python classes, and we gave an example of how the Geometry class converts the geometry supplied to JSON-LD. We then discussed the vocabulary creation and the involvement of the Protégé ontology editor in the creation process. After we delved into the arguments accepted by the tool and how to use them, followed by how the tool validates the input and tests the output. The next chapter will demonstrate how the tool works on a simple CityJSON file.

6 Demonstration

This chapter will demonstrate the tool using a simple CityJSON example from the CityJSON website. The file is called cube.city.json, and it contains the following:

- Eight vertices.

- One city object of type "GenericCityObject", which in turn contains:

  * A geometry of type "Solid".

  * LoD equals to "1".

- An attribute in the form of a JSON object.

- A Metadata object including the geographical extent.

- A Transform object containing a Scale and a Translate object.

The city object is supposed to represent a unit cube, hence the name cube.city.json, as seen in Figure 27.
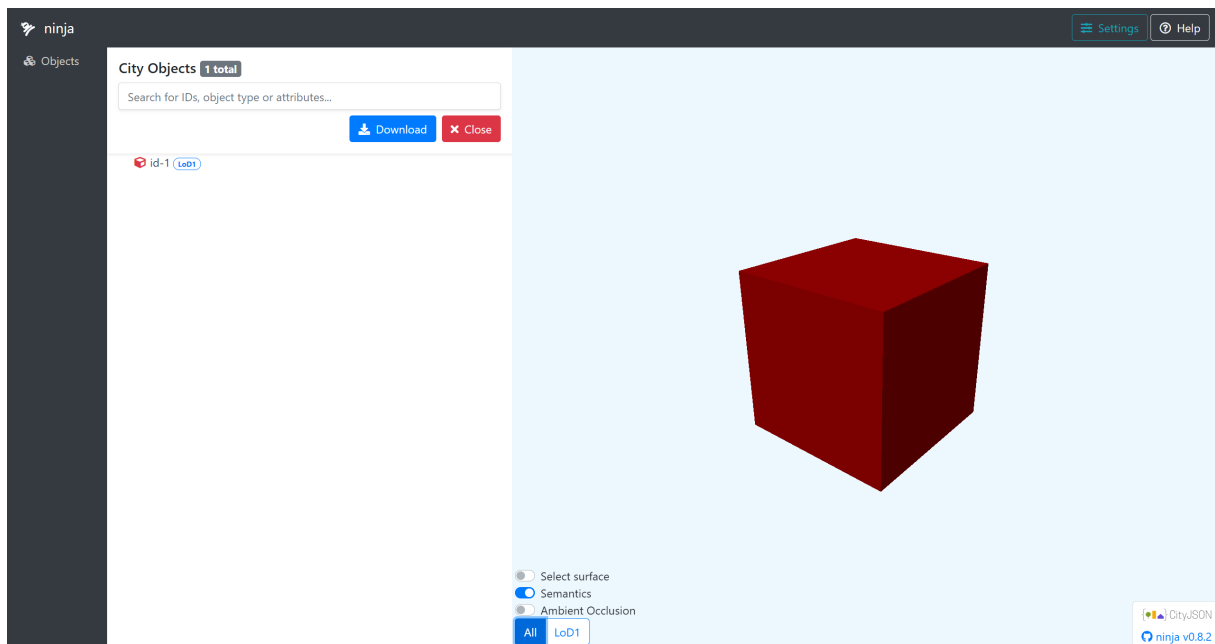
Figure 27: Visualization of cube.city.json using CityJSON Ninja

The cube.city.json, depicted in Figure 28, was converted to cube.city.jsonld shown in Figure 29, using the following command:

python main.py -i cube.city.json -o cube.city.jsonld -b http://example.com/ -id eaeceeaa-3f66-429a-b81d-bbc6140b8c1cclea -f

```json
{

    "CityObjects": {
        "id-1": {
            "geometry": [
                {
                    "boundaries": [

                        [
                            [ [0, 1, 2, 3] ],
                            [ [4, 5, 1, 0] ],
                            [ [5, 6, 2, 1] ],
                            [ [6, 7, 3, 2] ],
                            [ [7, 4, 0, 3] ],
                            [ [7, 6, 5, 4] ]
                        ]

                    ],
                    "lod": "1",
                    "type": "Solid"
                }
            ],

            "attributes": {
                "function": "something"
            },

            "type": "GenericCityObject"
        }
    },
    "type": "CityJSON",
    "version": "2.0",
    "vertices":
    [
        [0 , 0, 1000],
        [1000 , 0, 1000],
        [1000 , 1000, 1000],
        [0 , 1000, 1000],
        [0 , 0, 0],
        [1000 , 0, 0],
        [1000 , 1000, 0],
        [0 , 1000, 0]
    ],
    "metadata": {
        "geographicalExtent": [0, 0, 0, 1, 1, 1]
    },
    "transform": {
        "scale": [0.001, 0.001, 0.001],
        "translate": [0, 0, 0]
    }

}
```

Figure 28: cube.city.json

```
{
    "@context": {
        "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#", "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
        "cj": "https://www.cityjson.org/ont/cityjson.ttl#", "xsd": "http://www.w3.org/2001/XMLSchema#",
        "geosparql": "http://www.opengis.net/ont/geosparql#", "ex": "http://example.com/"
    },
    "@id": "http://example.com/eaeceeaa-3f66-429a-b81d-bbc6140b8c1cclea", "@type": "cj:CityJSON",
    "cj:type": "CityJSON", "cj:version": "2.0",
    "cj:hasVertices": {
        "@list": [
            {   "@type": "cj:Vertex",
                "cj:vertexX": { "@value": 0, "@type": "xsd:integer" }, "cj:vertexY": { "@value": 0,  "@type": "xsd:integer" },
                "cj:vertexZ": { "@value": 1000, "@type": "xsd:integer" }
            },
            {   "@type": "cj:Vertex",
                "cj:vertexX": { "@value": 1000, "@type": "xsd:integer" }, "cj:vertexY": { "@value": 0,  "@type": "xsd:integer" },
                "cj:vertexZ": { "@value": 1000, "@type": "xsd:integer" }
            },
            {   "@type": "cj:Vertex",
                "cj:vertexX": { "@value": 1000, "@type": "xsd:integer" },  "cj:vertexY": { "@value": 1000,  "@type": "xsd:integer" },
                "cj:vertexZ": { "@value": 1000, "@type": "xsd:integer" }
            },
            {   "@type": "cj:Vertex",
                "cj:vertexX": { "@value": 0, "@type": "xsd:integer" }, "cj:vertexY": { "@value": 1000,  "@type": "xsd:integer" },
                "cj:vertexZ": { "@value": 1000, "@type": "xsd:integer" }
            },
            ...
        ]
    },
    "cj:hasCityObjects": [
        {   "@id": "ex:id-1",   "@type": "cj:FirstLevelCityObject", "cj:type": "GenericCityObject", "cj:hasAttribute": { "function":
        ↪   "something" },
            "cj:hasGeometry": [
                {   "@type": "cj:Geometry",  "cj:type": "Solid", "cj:lod": "1",
                    "geosparql:asWKT": { "@value": "MULTIPOLYGON (((0 0, 1 0, 1 1, 0 1, 0 0)), ((0 1, 1 1, 1 0, 0 0, 0 1)))",  "@type":
                    ↪   "geosparql:wktLiteral" },
                    "cj:hasBoundingBox": {
                        "@type": "cj:Solid",
                        " cj:hasExteriorShell": {
                            "@type": "cj:ExteriorShell",
                            "cj:hasSurface": [
                                {   "@type": "cj:Surface",
                                    "cj:hasExteriorBoundary": {
                                        "@type": "cj:ExteriorShell",
                                        "cj:hasLineString": [
                                            {   "@type": "cj:LineString",
                                                "cj:hasPoint": [
                                                    {   "@type": "cj:Point",
                                                        "cj:boundaryX": {  "@value": 0, "@type": "xsd:float" },  "cj:boundaryY": {
                                                        ↪   "@value": 0, "@type": "xsd:float" }, "cj:boundaryZ": {  "@value": 1, "@type":
                                                        ↪   "xsd:float" }},
                                                    {   "@type": "cj:Point",
                                                        "cj:boundaryX": {  "@value": 1, "@type": "xsd:float" }, "cj:boundaryY": {  "@value":
                                                        ↪   0, "@type": "xsd:float" }, "cj:boundaryZ": {  "@value": 1, "@type": "xsd:float"
                                                        ↪   }},
                                                    {   "@type": "cj:Point",
                                                        "cj:boundaryX": {  "@value": 1, "@type": "xsd:float" }, "cj:boundaryY": {  "@value":
                                                        ↪   1, "@type": "xsd:float" }, "cj:boundaryZ": {  "@value": 1, "@type": "xsd:float"
                                                        ↪   }},
                                                    ...
    ]}]}]}]}}}]}],
    "cj:hasTransform": {
        "@type": "cj:Transform",
        "cj:hasScale": {   "@type": "cj:Scale",
                        "cj:scaleX": { "@value": 0.001, "@type": "xsd:float" }, "cj:scaleY": { "@value": 0.001, "@type": "xsd:float" },
                        "cj:scaleZ": { "@value": 0.001, "@type": "xsd:float" }
                    },
        "cj:hasTranslate": {   "@type": "cj:Translate",
                        "cj:translateX": {   "@value": 0,   "@type": "xsd:float" }, "cj:translateY": {   "@value": 0,   "@type":
                        ↪   "xsd:float" },
                        "cj:translateZ": {   "@value": 0,   "@type": "xsd:float" }
                    }
    },
    "cj:hasMetadata": {   "@type": "cj:Metadata",
                    "cj:hasGeographicalExtent": {
                        "@type": "cj:GeographicalExtent",
                        "cj:minX": { "@value": 0, "@type": "xsd:float"  }, "cj:maxX": { "@value": 1, "@type": "xsd:float"  },
                        "cj:minY": { "@value": 0, "@type": "xsd:float"  }, "cj:maxY": { "@value": 1, "@type": "xsd:float"  },
                        "cj:minZ": { "@value": 0, "@type": "xsd:float"  }, "cj:maxZ": { "@value": 0, "@type": "xsd:float"  }
                    }
    }
}
```

Figure 29: The output[4] after the conversion (cube.city.jsonld)

---

[4]The output has been truncated to fit the page. For the complete example, please visit this link:
https://github.com/aly1551995/CityJSON-LD/blob/main/Code/src/output/cube.city.jsonld

6.1 Fuseki

Now that we have a valid JSON-LD of the cube file, we can use all the functionalities that RDF permits, including the ability to query the data using SPARQL and its extension GeoSPARQL. For this purpose, we set up an Apache Jena Fuseki server. Fuseki is a SPARQL server that can run as a service, WAR, or standalone server. First, we ran Fuseki in a Docker container as a standalone server using the following command:

docker run -d –name fuseki-geo -p 3030:3030 ghcr.io/zazuko/fuseki-geosparql [5]

After the Docker container is up and running, we then created a new dataset using an arbitrary name in this case "Cube" with the TDB2 option for persistency, as seen in Figure 30.
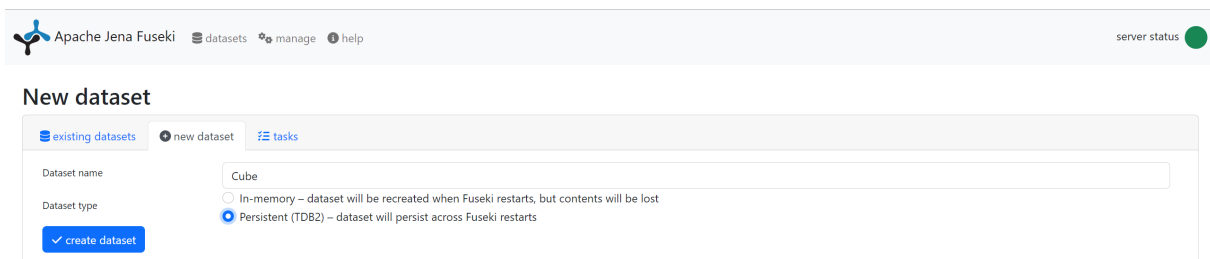
Figure 30: Adding a new dataset to Fuseki server

Then, we upload the file to the newly created dataset and get a confirmation of the successful upload in the form of the number of triples uploaded to the server. In this case, it is 243 triples, as shown in Figure 31.
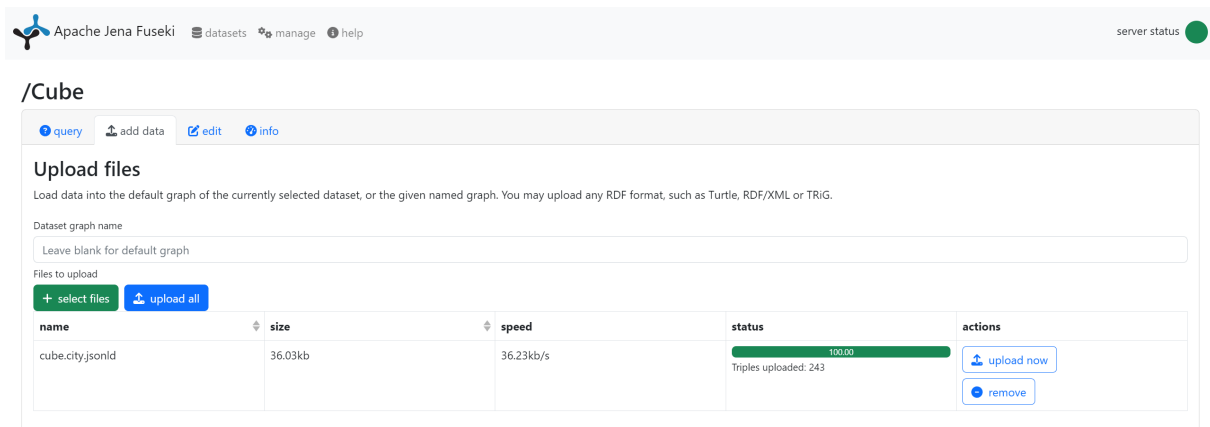
Figure 31: Uploading the tool's output to the Fuseki server

---

[5]ghcr.io/zazuko/fuseki-geosparql is an unofficial Docker image of Fuseki with GeoSPARQL installed.

6.2 SPARQL

Now that the data is uploaded, we can use SPARQL queries to query the triples, verify the information about the data mentioned in Section 6, and retrieve the new data created by the conversion tool. From now on, the structure for this section and Section 6.3 will consist of subsections demonstrating each query within it. Each query and its response will be explained and visualized. To avoid redundancy, we are omitting previously mentioned parts of queries.

6.2.1 Number of Vertices

First, we will check how many vertices the file contains. Figure 32 shows both the query and response. The query consists of the following:

- **PREFIX cj:** Defines a shorthand for the CityJSON ontology.

- **PREFIX rdf:** Defines a shorthand for the RDF syntax namespace.

- **SELECT (COUNT(?vertex) AS ?numVertices):** Selects the count of vertices and labels it as `?numVertices`.

- **WHERE ?city cj:hasVertices ?vertices:** Finds city objects and their associated vertex lists.

- **WHERE ?vertices rdf:rest*/rdf:first ?vertex:** Traverses the list of vertices to find each vertex using the RDF list structure.

Figure 32: Number of Vertices query and response

44

6.2.2 Number of CityObjects

The query is similar to the one used in Section 6.2.1 to find the number of vertices, except now we are looking for the number of city objects in the file, as shown in Figure 33. The query consists of the following:

- **SELECT (COUNT(?city) AS ?numCityObjects):** Selects the count of city objects and labels it as `?numCityObjects`.

- **WHERE ?city cj:hasCityObjects ?numCityObjects:** Finds city objects within a city.

Figure 33: Number of city objects in the file query and response

6.2.3 CityObjects and their LoDs

Below is the SPARQL query to retrieve the Level of Detail (LoD) of city geometries, as shown in Figure 34, along with a detailed breakdown of each part:

- **SELECT ?cityObjects ?lod:** Selects the city objects and their Level of Detail (LoD).

- **WHERE ?city cj:hasCityObjects ?cityObjects**: Finds city objects within a city.

- **WHERE ?cityObjects cj:hasGeometry ?geometry**: Finds city objects and their associated geometries.

- **WHERE ?geometry cj:lod ?lod**: Retrieves the geometries' Level of Detail (LoD).

45

/Cube

Figure 34: City objects and their LODs query and response

6.2.4 Retrieve WKT strings of CityObjects

The query retrieves the WKT string of each city object geometry. A city object may contain zero or more geometry with different LoDs. Hence, there is a possibility of yielding no data or the same city object more than once. Therefore, the addition of the LoD to differentiate between the WKT strings of the same city object with different geometries. This query marks the first time we have used the GeoSPARQL vocabulary in our queries. The query uses the vocabulary to retrieve the WKT data property. We will elaborate more on GeoSPARQL and its functions in Section 6.3. Below is the SPARQL query to retrieve WKT strings of geometries, along with a detailed breakdown of each part:

- **PREFIX geosparql:** Defines a shorthand for the GeoSPARQL ontology namespace.

- **SELECT ?cityObjects ?lod ?wkt:** Selects the variables representing city objects, levels of detail (LoD), and the Well-Known Text (WKT) representation of geometries.

- **WHERE ?geometry geosparql:asWKT ?wkt**: Retrieves the WKT representation of the geometry.

46

/Cube

Figure 35: WKT strings of city objects query and response

6.2.5 Retrieve Metadata's geographical extent

This SPARQL query retrieves the geographical extent found in the Metadata object in a CityJSON file. This query has the potential to return with no data as the Metadata object is optional in CityJSON, and even if it is present, the geographical extent object is also optional; the details of the query are as follows:

- **SELECT ?minX ?maxX ?minY ?maxY ?minZ ?maxZ:** Selects the minimum and maximum X, Y, and Z coordinates.

- **WHERE ?city cj:hasMetadata ?metadata:** Finds city objects and their associated metadata.

- **WHERE ?metadata cj:hasGeographicalExtent ?geographicalExtent:** Finds the geographical extent within the Metadata.

- **WHERE ?geographicalExtent cj:minX ?minX ; cj:maxX ?maxX ; cj:minY ?minY ; cj:maxY ?maxY ; cj:minZ ?minZ ; cj:maxZ ?maxZ:** Retrieves the minimum and maximum X, Y, and Z coordinates of the geographical extent.

47

/Cube

Figure 36: Metadata's geographical extent query and response

6.3   GeoSPARQL

In this section, we will demonstrate some GeoSPARQL queries applied to the tool's output. The section will follow the previous structure as Section 6.2 with one addition: the use of Yasgui, a tool to visualize SPARQL and GeoSPARQL queries. We give it the SPARQL endpoint URL, and it connects to it. We can run the queries from it, and if there is a WKT string, it displays it on 2D and 3D maps.

6.3.1   Calculate the Convex Hull

In this GeoSPARQL query, we take advantage of one of GeoSPARQL Non-Topological Functions, which is the **convexHull** function that takes as an input a WKT string representing the geometry of the object and returns an object containing all the points in the convex hull of the input of object. Below is the formulated GeoSPARQL query to calculate the convex hull of the geometry of the city objects, as shown in Figure 37, and in Figure 38, we can see the visualization of the response. Next, we will break down the query in detail:

- **PREFIX geof: http://www.opengis.net/def/function/geosparql/:** Defines the prefix geof to refer to the GeoSPARQL function namespace.

- **SELECT ?cityObject (geof (?geometryWKT) AS ?convexHull):** Selects the city objects and calculates the convex hull of their geometries, labeling it as ?convexHull.

/Cube

Figure 37: The calculated convex hull

Figure 38: Visualization of the calculated convex hull (convex hull in red)

6.3.2 Calculate the Boundary

This GeoSPARQL query uses the Non-Topological function **boundary**, a function that takes the geometry of an object as an input and returns the boundary surrounding that object. Figure 39 shows the GeoSPARQL query used to calculate the boundary of the city objects' geometry. Figure 40 shows the response's visualization. The query consists of the following:

- **SELECT?cityObject ?geometryWKT (geof:boundary(?geometryWKT) AS ?boundary):** Selects the city objects, their geometries in WKT format, and calculates the boundary of the geometries, labeling it as ?boundary.

Figure 39: The Calculated Boundary

```
     Query ×      +

⚙    http://localhost:3031/Cube/sparql

1 ▾ PREFIX cj: <https://www.cityjson.org/ont/cityjson.ttl#>
2    PREFIX geosparql: <http://www.opengis.net/ont/geosparql#>
3    PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
4
5    SELECT ?cityObject ?geometryWKT ?boundary ?boundaryColor ?geometryWKTColor
6 ▾  WHERE {
7        ?cityObject cj:hasGeometry ?geometry .
8        ?geometry geosparql:asWKT ?geometryWKT .
9
10       # Calcualte the boundary and bind it to the ?boundary variable
11       BIND(geof:boundary(?geometryWKT) AS ?boundary)
12       # Bind the color BLACK to the ?boundary variable
13       BIND("#000000" AS ?boundaryColor)
14       # Bind the color WHITE to the ?geometryWKTColor variable
```

| ▦ Table | ≡ Response | 🖼 Gallery | 📊 Chart | 📍 Geo | 🌐 Geo-3D | 📊 Geo events | 📄 Markup | ⌁ Network | ▦ Pivot | ≡ Timeline | 1 result in 0.011 seconds |

Figure 40: Visualization of the calculated boundary (boundary in black)

6.4   Demonstration Summary

In the chapter, we showcased the conversion tool and highlighted its advantage in enabling SPARQL and GeoSPARQL queries[6]. on the converted data. We specifically demonstrated how to upload the converted data into Apache Jena Fuseki and query it using SPARQL and GeoSPARQL. Some of the SPARQL queries we demonstrated were retrieving the number of vertices, the number of city objects, the LoDs of each city object, the WKT string of the city object's geometry, and finally, the retrieval of the geographical extent of the Metadata embedded in the CityJSON file. For GeoSPARQL, we showed two queries: one to calculate the convex hull and the other to calculate the boundary of an object. In the next chapter, we will showcase the tool's capabilities and the power of GeoSPARQL on actual city data, such as New York City and Helsinki.

---

[6]All the queries and commands used in this chapter can be found in this link https://github.com/aly1551995/CityJSON-LD/tree/main/Demo

7 Case Studies

In this Chapter, we will study two datasets of two cities: Helsinki and New York City. Both datasets have different characteristics. Both have been converted using the tool and uploaded to Fuseki using the same procedure shown in Chapter 6. Since the datasets represent real cities, utilizing more advanced GeoSPARQL queries is possible. Moreover, it allows us to emphasize the advantages of converting to JSON-LD and extending the features of CityJSON with WKT string for geometries by showcasing real-life situations and examples.

7.1 Helsinki

Before we delve into more advanced queries, we ran some queries to explore and get acquainted with the dataset; also, it will help benchmark this dataset with the New York dataset, as we will see in Section 7.3. Here is a summary of some of the properties of the Helsinki dataset:

Metric	Value
Number of Triples	1,911,692
Number of City Objects	677
Number of Buildings	677
Number of Vertices	72,519
Types of Geometry	Solid
Types of LoDs	LoD1 and LoD2
Coordinate Reference System (CRS)	EPSG 3879

Table 2: Summary of the Helsinki Dataset.

7.1.1 Intersection

First, we will take a building in Helsinki, shown in Figure 41, and check whether it intersects with any other building. The query, shown in Figure 42, first retrieves the WKT of the target building (BID_9c1fc3d5c69c4de1b61ace84432f72f0). Since this dataset contains two LoD geometry representations, we will pick *LoD1*. Simpler geometry is better for faster response due to less complexity and does not require the *boundary* function. As we will see in Section 7.2.1, we get all the WKT of the other buildings with *LoD1*. We filter based on whether they intersect with the target building or not using the GeoSPARQL function *sfIntersects*, which is a function when given two geometries return a true if they intersect and false if they do not intersect; we then exclude the geometry of the target building itself and then calculate the intersection of geometry of the target building with the other building using the GeoSPARQL *intersection* function which given two geometries return all points in the intersection of the input geometries. As we can see again in Figure 42, two buildings intersect with the target building, and we have the WKT of the intersections shown in Figure 43.

Figure 41: The target building (BID_9c1fc3d5c69c4de1b61ace84432f72f0).

Figure 42: GeoSPARQL query and response for buildings that intersect with BID_9c1fc3d5c69c4de1b61ace84432f72f0.
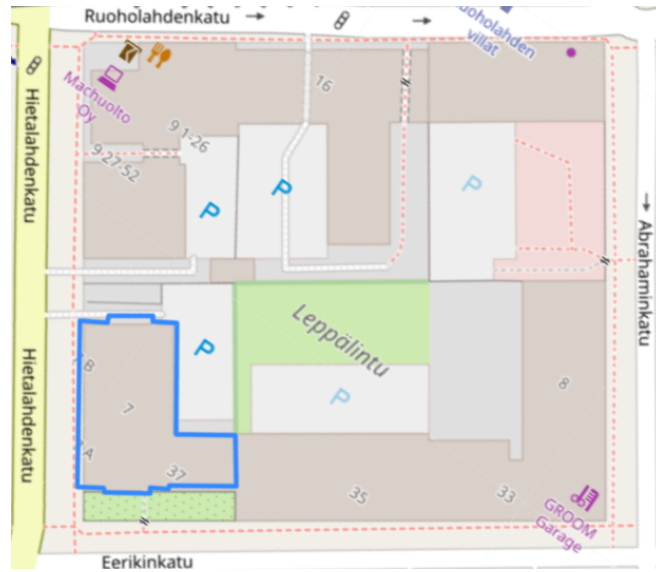
(a) Left Intersecting Building.

(b) Intersection points.
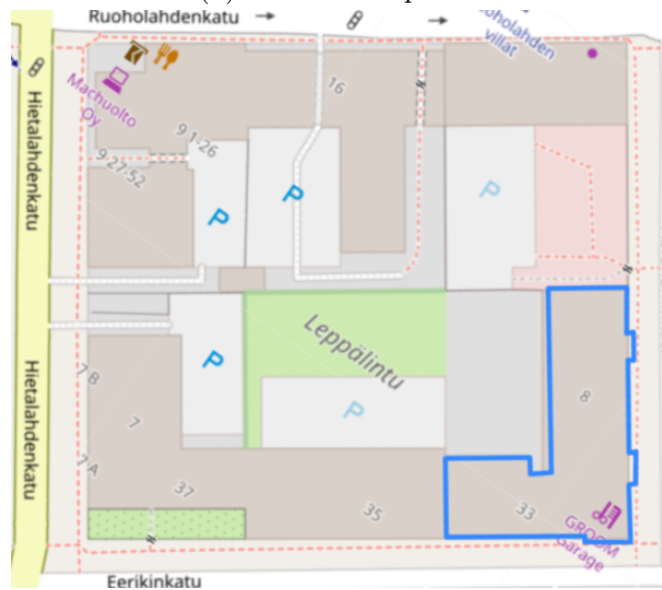
(c) Right Intersecting Building.

Figure 43: Intersection points and the intersecting buildings.

We can even take this a step further and get all the buildings that intersect with one another using the query in Figure 44. The query checks if a building intersects with another building, and if it does, it returns the intersection point. We can show all the intersection points by modifying the query, as seen in Figure 45. We can also show a map of all the buildings that intersect in the city, as seen in Figure 46.

Figure 44: GeoSPARQL query and response for all intersection points in the city.
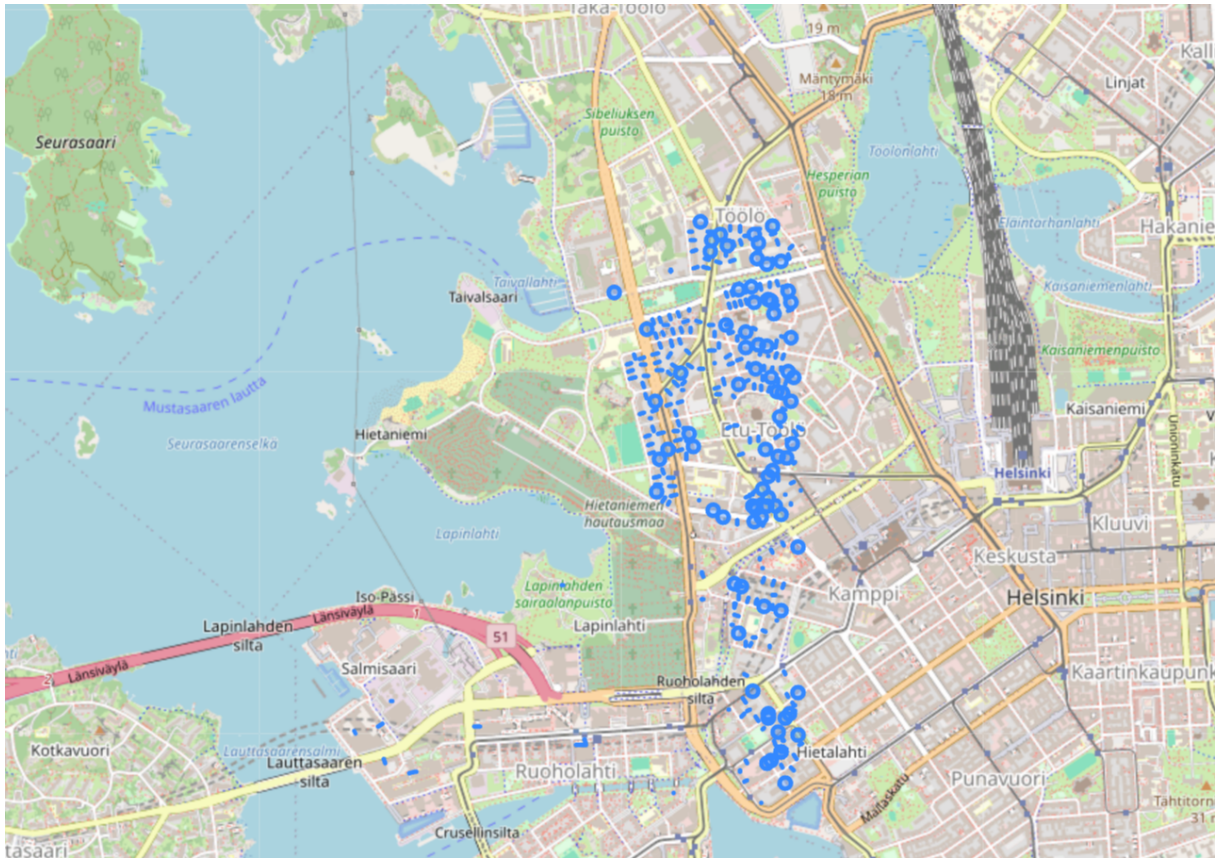
Figure 45: Visualization of all the intersection points.
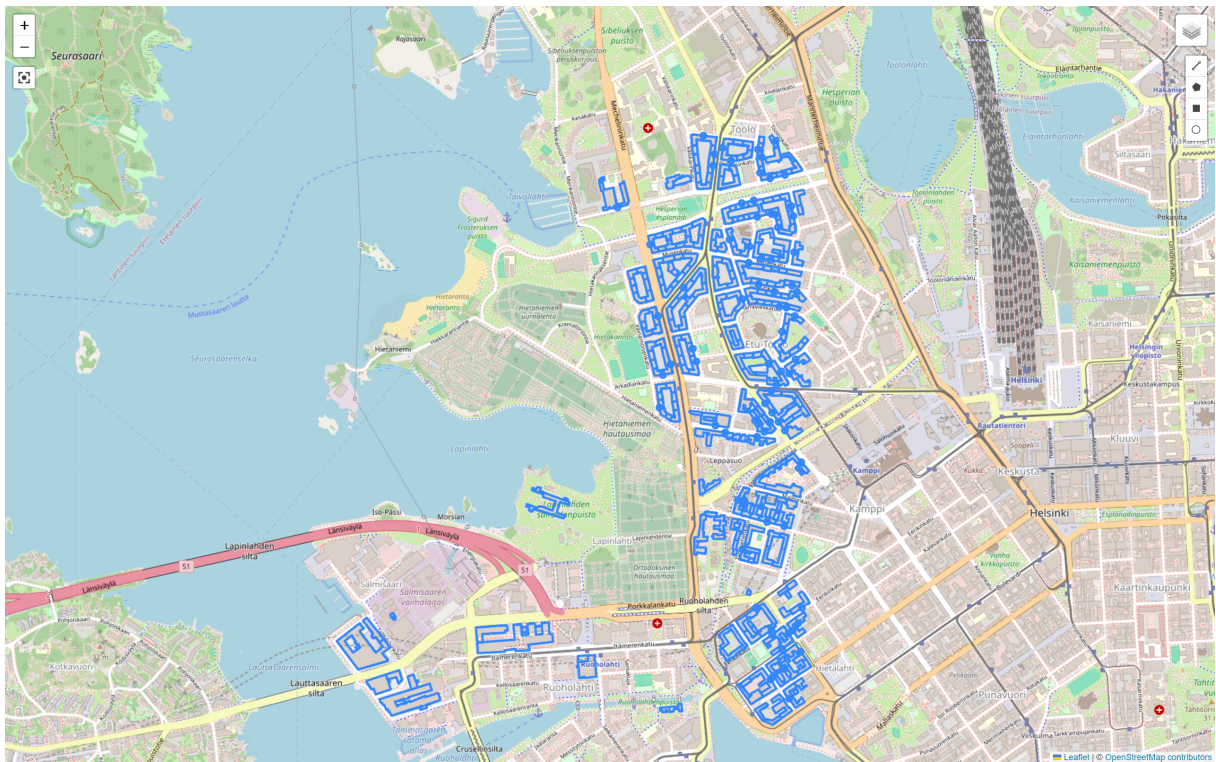
Figure 46: Visualization of all the buildings that have intersection points.

7.1.2 Union

Now, another query, as seen in Figure 47, demonstrates that GeoSPARQL can be used to combine all the buildings from Section 7.1.1 into a single WKT for visualization purposes. This can be achieved using the *union* function, which, when given two geometries, returns the union of those geometries' points. However, it is not explicitly stated whether the GeoSPARQL *union* function merges intersecting points during the union, similar to what the PostGIS union function does, as discussed in this link [7]. The result, as shown in Figure 48, suggests that the function does indeed merge the intersections. Given that we have the IRIs of the buildings from the previous query's results, we can retrieve their WKT individually and then apply the union function on them.

Figure 47: GeoSPARQL union query and response for the three buildings mentioned in Section 7.1.1.

[7] https://postgis.net/workshops/uk/postgis-intro/geometry_returning.html#st-union
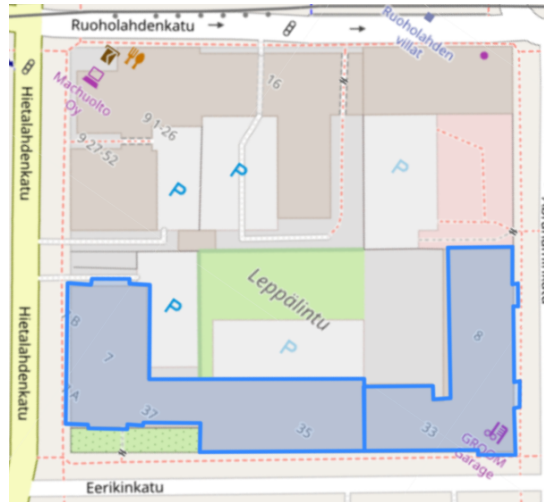
Figure 48: Visualization of the union of the three buildings mentioned in Section 7.1.1.

7.1.3 Concatenation

We can also use the query shown in Figure 49 to concatenate all the geometries to visualize all the buildings in the city with specific LoDs. This query could help visualize a neighborhood or district, as Figures 50 and 51 show.

Figure 49: GeoSPARQL concatenation query and response for all buildings with LoD1.

Figure 50: Visualization of the concatenation of all the building's geometries with LoD1.
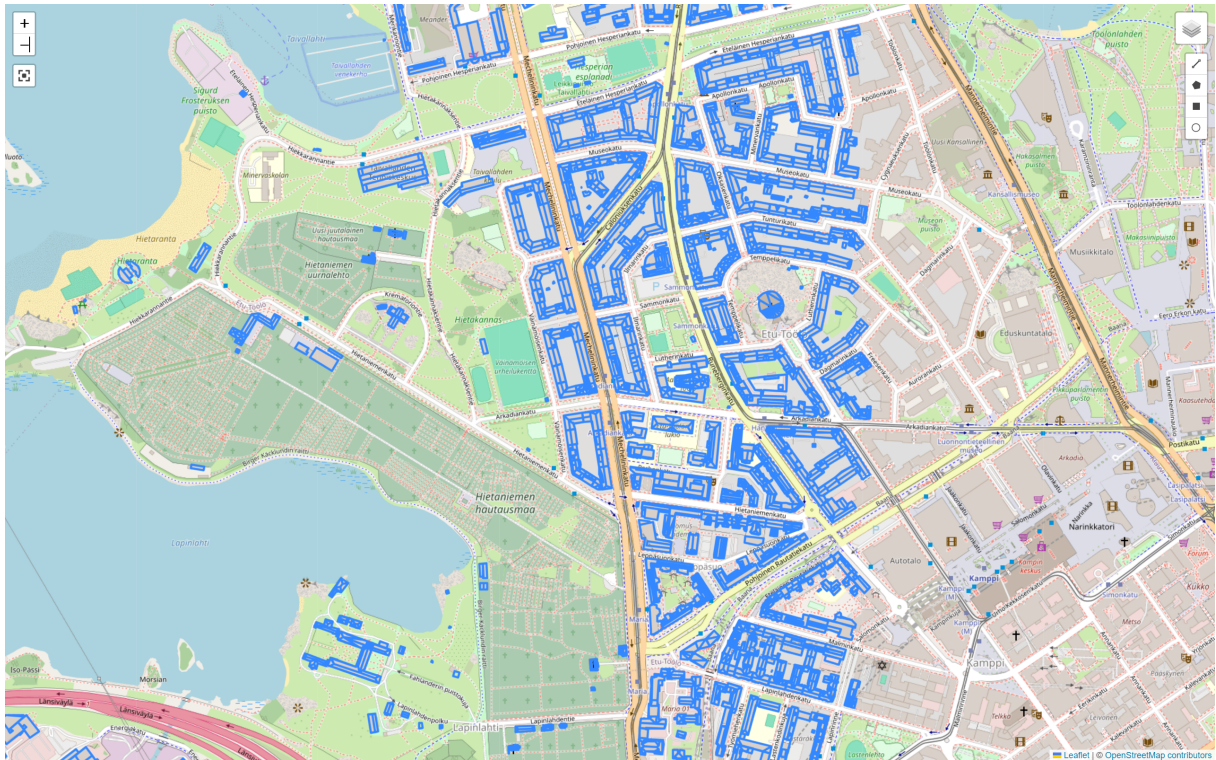
Figure 51: Visualization of the concatenation of all the building's geometries with LoD2.

7.1.4 Radius

Assuming we want to form a perimeter around the same building from the previous Sections 7.1.1 and 7.1.2, then retrieve all the buildings within that perimeter, this can be applied using the query shown in Figure 52. The query first retrieves the WKT of the building and then applies the GeoSPARQL *buffer* function. This function takes a geometry, a radius, and a unit of measure as inputs. It outputs a polygon of points with a distance less than or equal to the given radius from the geometry's center. Now the issue is that the buffer function seems to have problems, as indicated by an issue on the official GeoSPARQL GitHub repository, which states that the unit of measure can be problematic; in our case, if we use meters as a unit, we receive an empty output even thought it should return a radius of 1 KM. However, it seems to work when using the degree as the unit. That is why we have a degree as the unit of measure in the query with a value not equivalent to 1 KM in degrees. However, we verified the correctness of the output, as shown in Figure 54, by drawing a circle of a 1 KM radius from the address of the building using the online tool Map Developers, and the output matches the result shown in Figure 53. Finally, we get all the buildings and then filter using the GeoSPARQL *sfWithin* to filter on the buildings within that created boundary.

Figure 52: The perimeter around the building previously mentioned in Sections 7.1.1 and 7.1.2 and the buildings within that perimeter.

Figure 53: Visualization of the perimeter around the building previously mentioned in Sections 7.1.1 and 7.1.2 and the buildings within that perimeter.

Figure 54: Verification of the 1 KM radius output of the GeoSPARQL query.

7.1.5 Heights of Buildings

We noticed that all the buildings in the Helsinki dataset contain the *measuredHeight* attribute, which indicates the height of the building. This attribute is not standardized. It is just a deduction from the analysis of the data. Hence, it is only applicable to this dataset. So, we can use this attribute to retrieve each building's height. The difficulty is that the CityJSON attribute field can be any valid JSON object. Therefore, we would need to either string manipulate the JSON object to retrieve the needed values, which is possible within the query itself, as shown in Figure 55, or post-process the query result using Python to obtain the required values, as shown in Figure 56.

Figure 55: GeoSPARQL query and response to retrieve *measuredHeight* attribute using string manipulation.

```python
1    import json
2    from SPARQLWrapper import SPARQLWrapper, JSON
3
4    # Set up the SPARQL endpoint
5    sparql = SPARQLWrapper("http://localhost:3031/Helsinki/sparql")
6
7    # Define the SPARQL query to fetch the attributes along with the city object
8    query = """
9    PREFIX cj: <https://www.cityjson.org/ont/cityjson.ttl#>
10
11   SELECT ?cityObject ?attributes
12   WHERE {
13       ?cityObject cj:hasAttribute ?attributes .
14   }
15   """
16
17   # Set the query
18   sparql.setQuery(query)
19   sparql.setReturnFormat(JSON)
20
21   # Execute the query and get the results
22   try:
23       results = sparql.query().convert()
24   except Exception as e:
25       print(f"Error querying the SPARQL endpoint: {e}")
26       results = None
27
28   # Initialize variables for calculating the average height
29   total_height = 0
30   count = 0
31   details = []
32
33   # First pass: Calculate the total height and count valid measurements
34   if results and "results" in results and "bindings" in results["results"]:
35       for result in results["results"]["bindings"]:
36           city_object = result["cityObject"]["value"]
37           attributes_values = json.loads(result["attributes"]["value"])
38
39           # Check if "measuredHeight" is in the attributes and is a number
40           measuredHeight = attributes_values.get("measuredHeight", None)
41           if measuredHeight is not None:
42               try:
43                   measuredHeight = float(measuredHeight)
44                   total_height += measuredHeight
45                   count += 1
46               except ValueError:
47                   measuredHeight = "N/A"
48
49           # Store the details for the second pass
50           details.append((city_object, measuredHeight))
51
52       # Calculate the average height if any valid heights were found
53       if count > 0:
54           average_height = total_height / count
55       else:
56           average_height = 0.0
57
58       # Print the summary at the beginning
59       print(f"Query returned {count} results with valid heights. Average Height: {average_height:.7f}.")
60
61       # Second pass: Print each city's details
62       for city_object, measuredHeight in details:
63           print(f"City Object: {city_object}, measuredHeight: {measuredHeight}")
64   else:
65       print("No results found or an error occurred.")
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
Query returned 677 results with valid heights. Average Height: 17.0015362.
City Object: http://example.com/BID_00c42d1b-0578-4302-8ffc-85d398e201ee, measuredHeight: 2.51
City Object: http://example.com/BID_01a54718-5fc7-4d3d-960d-f724addd29d0, measuredHeight: 21.91
City Object: http://example.com/BID_01c22dce-474a-45d4-b7aa-3d51aab08382, measuredHeight: 19.97
City Object: http://example.com/BID_01e326b6-98b7-423a-ae37-5832cff14e08, measuredHeight: 3.0
City Object: http://example.com/BID_01f4c238-726b-4af8-ba92-12fd88452cc2, measuredHeight: 23.5
City Object: http://example.com/BID_03052758-a54f-46ec-a1c3-37f410b1e3d1, measuredHeight: 4.06
City Object: http://example.com/BID_030c234e-ed21-4b13-b941-bb48423d2239, measuredHeight: 9.0
```

Figure 56: Using Python to query GeoSPARQL endpoint and extract *measuredHeight*.

63

Subsequently, we can measure the average height of all the buildings in the dataset, as seen in Figure 57.

Figure 57: GeoSPARQL query and response to retrieve the average *measuredHeight*.

7.2 New York City

Similar to the approach we used in Section 7.1, here is a summary of the New York City file:

Metric	Value
Number of Triples	6,287,270
Number of City Objects	23,777
Number of Buildings	23,777
Number of Vertices	1,035,804
Types of Geometry	MultiSurface
Types of LoDs	LoD2
Coordinate Reference System (CRS)	EPSG 2263

Table 3: Summary of the New York City Dataset.

7.2.1 Intersection

The New York City dataset is characterized by buildings with LoD2 only, meaning they are buildings with complex geometry and might have repeated and overlapping coordinates. So, in order for the GeoSPARQL *intersection* function to work correctly, we need to simplify the geometry, and that is achieved by calculating the boundary of the buildings and comparing it with the other buildings rather than what we did in Section 7.2.1 where we just first retrieved the WKT and then filtered on buildings that intersect and then computed the intersection on the retrieved WKT string directly, However, in this case, we also have to use the GeoSPARQL *boundary* function, which is a function given a geometry returns the closure of the boundary of the specified geometry, The query, response, and the visualization are shown in Figures 58 and 59.

Figure 58: GeoSPARQL query and response for buildings that intersect with the target building.
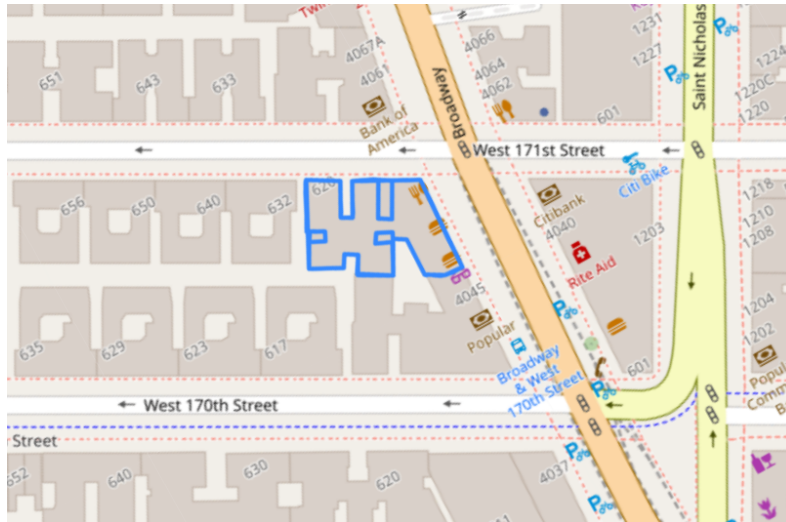
(a) Target Building.

(b) Intersection points.
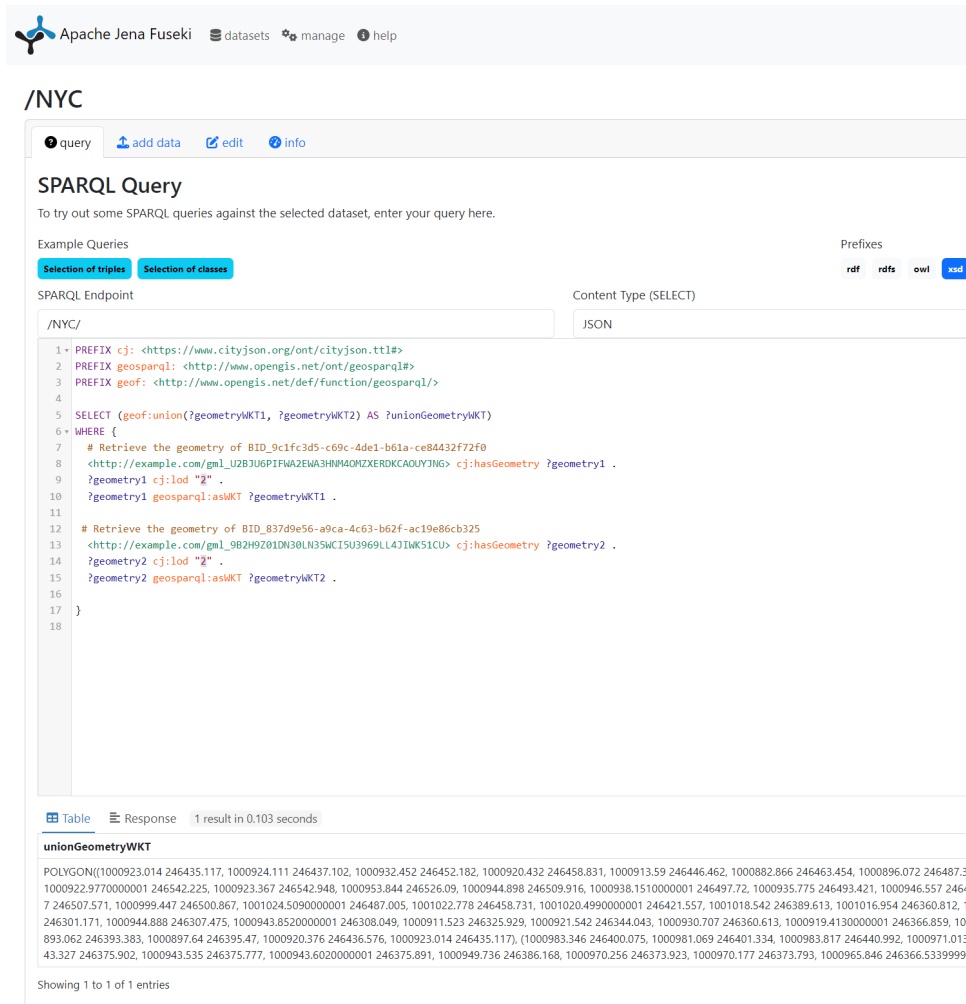
(c) Intersecting Building.

Figure 59: Target building and the intersecting building.

66

7.2.2 Union

The below query is the same as the one executed in Section 7.1.2, with the difference of using a LoD2 building from the New York City dataset, the result as seen in Figures 60 and 61.

Figure 60: GeoSPARQL union query and response for the buildings mentioned in Section 7.2.1.

Figure 61: Visualization of the union of the buildings mentioned in Section 7.2.1.

7.2.3 Concatenation

This query required the use of SPARQLWrapper, a Python package that wraps around a SPARQL service to execute RDF queries remotely. Its usage was due to browser interface crashing because of the amount of data sent in the HTTP response from the Fuseki server. It also required the use of GeoPandas and Folium for conversion and visualization. In Figure 62, the Python script ran the query and generated a visualization of all the buildings in New York City, as seen in Figures 63 and 64.

```python
1   from SPARQLWrapper import SPARQLWrapper, JSON
2   import geopandas as gpd
3   from shapely import wkt
4   import folium
5
6   # Set up the SPARQL endpoint
7   sparql = SPARQLWrapper("http://localhost:3031/NYC/sparql")
8   # Define the SPARQL query to fetch the geometry collection with non-empty WKTs
9   query = """
10  PREFIX cj: <https://www.cityjson.org/ont/cityjson.ttl#>
11  PREFIX geosparql: <http://www.opengis.net/ont/geosparql#>
12  PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
13
14  SELECT (CONCAT("GEOMETRYCOLLECTION(", GROUP_CONCAT(?geometryWKT; separator=","), ")") AS ?geometryCollectionWKT)
15  WHERE {
16    {
17      SELECT ?geometry
18      WHERE {
19        # Retrieve the geometries of buildings
20        ?building cj:hasGeometry ?geometry .
21        ?geometry cj:lod "2" .
22      }
23    }
24    ?geometry geosparql:asWKT ?geometryWKT .
25  }
26
27  """
28  # Set the query
29  sparql.setQuery(query)
30  sparql.setReturnFormat(JSON)
31  # Execute the query and get the results
32  try:
33      results = sparql.query().convert()
34  except Exception as e:
35      print(f"Error querying the SPARQL endpoint: {e}")
36      results = None
37  # Check if results were returned
38  if results and "results" in results and "bindings" in results["results"]:
39      for result in results["results"]["bindings"]:
40          geometry_collection_wkt = result["geometryCollectionWKT"]["value"]
41
42          # Save the result to a file
43          with open("geometry_collection.wkt", "w") as file:
44              file.write(geometry_collection_wkt)
45              print("Geometry collection WKT saved to geometry_collection.wkt")
46
47          # Convert WKT to a Shapely geometry
48          geometry = wkt.loads(geometry_collection_wkt)
49
50          # Create a GeoDataFrame with EPSG:2263
51          gdf = gpd.GeoDataFrame(index=[0], crs="EPSG:2263", geometry=[geometry])
52
53          # Reproject to EPSG:4326 for visualization on a real-world map
54          gdf = gdf.to_crs(epsg=4326)
55
56          # Get the centroid of the geometry to center the map
57          centroid = gdf.geometry.centroid.iloc[0]
58
59          # Create a folium map centered around the geometry
60          m = folium.Map(location=[centroid.y, centroid.x], zoom_start=12)
61
62          # Add the geometry to the map
63          folium.GeoJson(gdf).add_to(m)
64
65          # Save or display the map
66          m.save("map.html")
67          print("Map saved to map.html")
68
69  else:
70      print("No results found or an error occurred.")
71
```

Figure 62: Python concatenation query, response, and visualization script for New York City dataset.
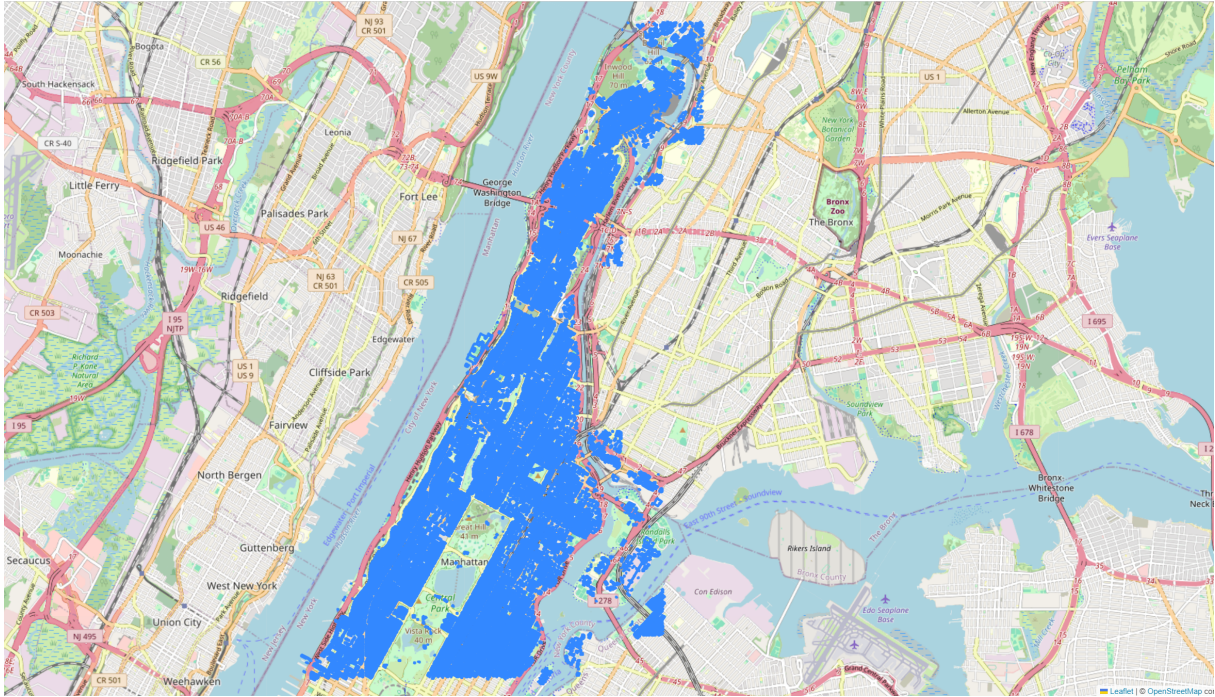
Figure 63: Visualization of the concatenation of all the buildings in New York City dataset.
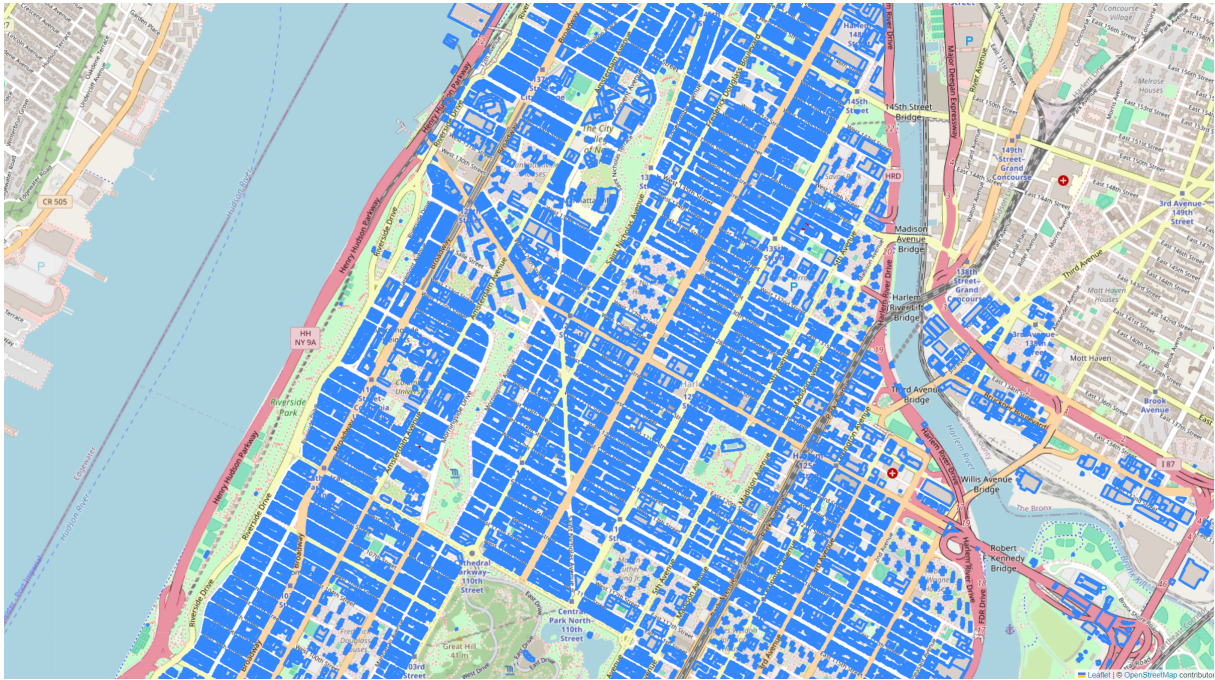
Figure 64: Visualization of the concatenation of all the buildings in New York City Dataset (Close-up).

7.2.4   Radius

Figures 65 and 66 show the query, response, and visualization for applying a 1 KM radius around the target building used in Section 7.2.1.


```
1   PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>
2   PREFIX cj: <https://www.cityjson.org/ont/cityjson.ttl#>
3   PREFIX geosparql: <http://www.opengis.net/ont/geosparql#>
4   PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
5
6   # Concatenates the original geometry, its buffered version, and other buildings' geometries into a GEOMETRYCOLLECTION WKT string
7   SELECT (CONCAT("GEOMETRYCOLLECTION(", STR(?geometryWKT), ",", GROUP_CONCAT(STR(?otherGeometryWKT); separator=","), ",", STR(?bufferedGeometry), ")") AS
8   WHERE {
9       # Retrieve the geometry of a specific building identified by its URI
10      <http://example.com/gml_U2BJU6PIFWA2EWA3HNM4OMZXERDKCAOUYJNG> cj:hasGeometry ?targetGeometry .
11
12      # Filter the geometry to ensure it's at Level of Detail 2 (LOD2)
13      ?targetGeometry cj:lod "2" .
14
15      # Extract the Well-Known Text (WKT) representation of the geometry
16      ?targetGeometry geosparql:asWKT ?geometryWKT .
17
18      # Apply a buffer of 1000 units (1 kilometers) around the original geometry
19      BIND(geof:buffer(?geometryWKT, 1000, uom:degree) AS ?bufferedGeometry)
20
21      # Find other buildings within the buffered area
22      ?otherBuilding cj:hasGeometry ?otherGeometry .
23      ?otherGeometry geosparql:asWKT ?otherGeometryWKT .
24      ?otherGeometry cj:lod "2" .
25
26      # Filter buildings that intersect with the buffered area
27      FILTER(geof:sfWithin(?otherGeometryWKT, ?bufferedGeometry))
28
29  }
30  GROUP BY ?geometryWKT ?bufferedGeometry
31
```

Figure 65: The Query and response to retrieve the perimeter around the building previously mentioned in Sections 7.2.1 and 7.2.2 and the buildings within that perimeter.
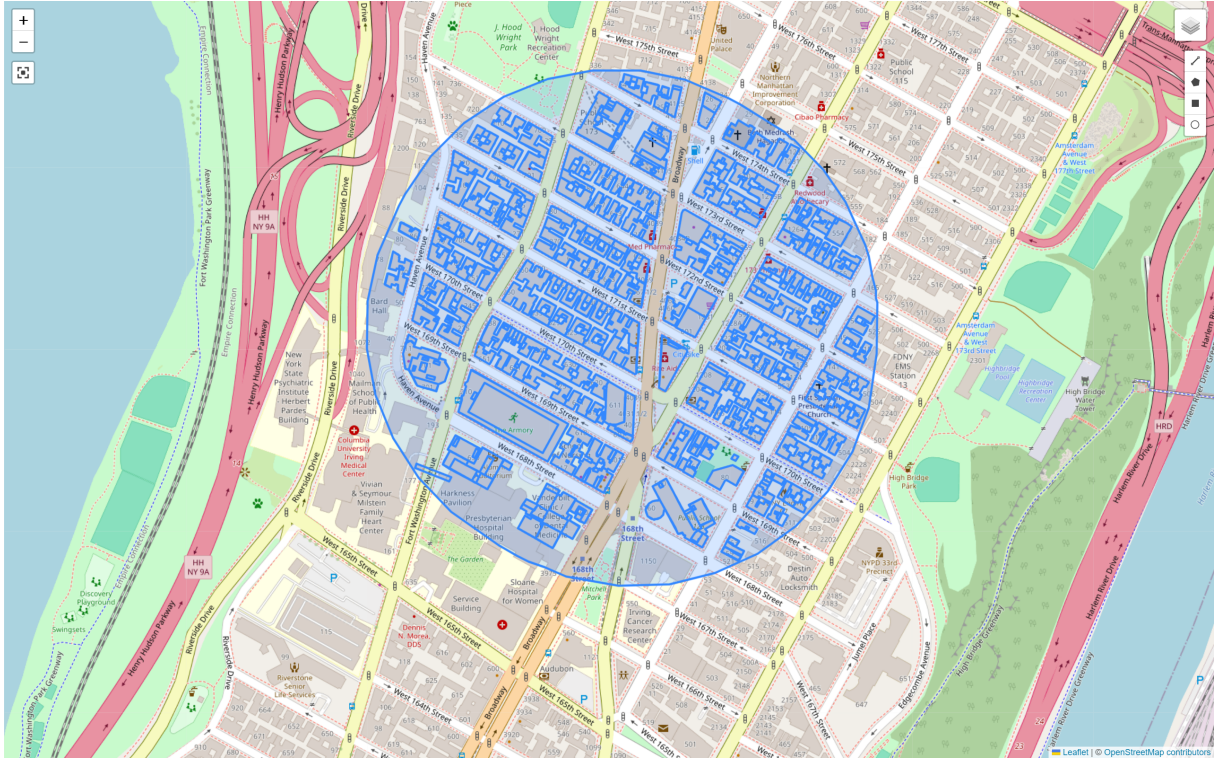
Figure 66: Visualization of the perimeter around the building previously mentioned in Sections 7.2.1 and 7.2.2 and the buildings within that perimeter.

7.3 Benchmark

This comparative case study of the Helsinki and New York City datasets highlights critical differences in dataset characteristics, conversion processes, query performance, and visualization methods. The analysis and benchmarking provide valuable insights into the strengths and challenges of handling datasets of varying complexity and size.

7.3.1 Dataset Characteristics

The Helsinki dataset, which contains 677 buildings and 72,519 vertices, contrasts sharply with the New York City dataset, featuring 23,777 buildings and over a million vertices. This significant difference in scale had a pronounced impact on processing times. The initial attempt to upload the New York dataset to Fuseki failed due to a "ran out of memory" error, caused by the large number of vertices overwhelming the system's capacity. To address this issue and successfully upload the data, the converted JSON-LD file was partitioned into six separate parts: one file containing all the data except the vertices, and five additional files, each containing a portion of the vertices. This partitioning allowed the upload process to proceed without exceeding memory limits. A summary of the characteristics of both datasets is shown in Table 4.

Metric	Helsinki	New York City
Coordinate Reference System (CRS)	EPSG 3879	EPSG 2263
File Size	20 MB	170 MB
Source	CityJSON	GitHub
Converted From CityGML	Yes	Yes
Time To Upload to Fuseki	≈ 4 mins	≈ 15 mins and required batching
Number of Vertices	72,519	1,035,804
Number of Attributes Per Building	12	3

Table 4: Characteristics Comparison between Helsinki and New York City Datasets.

7.3.2 Conversion Process

The conversion times varied according to the complexity of the datasets, with the Helsinki dataset converting faster than the New York dataset. Both datasets were converted from CityJSON to JSON-LD using the *cj2jld* tool, in both formatted and raw (unformatted) forms. The formatted conversion resulted in a file size that was at least twice as large as the unformatted version, indicating a significant increase in size due to formatting. Additionally, using PySHACL for validating the output proved impractical when the `--enable-pyshacl` argument was applied, as the validation process became excessively time-consuming. A summary of the conversion tool's performance with respect to the two datasets is provided in Table 5.

Metric	Helsinki	New York City
Conversion Time (without PySHACL)	≈ 26 seconds	≈ 1 minute and 4 seconds
Conversion Time (with PySHACL)	N/A	N/A
File Size After Conversion (Unformatted)	73.3 MB	326 MB
File Size After Conversion (Formatted)	318 MB	718 MB

Table 5: Conversion Tool Comparison between Helsinki and New York City Datasets.

7.3.3 Visualization

Visualization methods differed between datasets, with the Helsinki dataset visualization using more straightforward online tools like WKT MAP. Due to its size and complexity, the New York City dataset often required external tools like Python's *Folium* and *GeoPandas* for adequate visualization, especially when handling large queries like concatenation.

7.3.4 Query Performance

Query execution times were significantly shorter for the Helsinki dataset across all queries. The simpler geometry of Helsinki buildings enabled more efficient intersection, union, and concatenation operations. In contrast, the New York dataset required more computational resources due to its LoD2 complexity and the necessity of additional GeoSPARQL functions, such as *boundary*, to simplify the geometry. A summary of the query response times is provided in Table 6.

Query	Helsinki (Time)	New York City (Time)
Intersection	0.203 seconds	1.818 seconds
Union	0.071 seconds (3 buildings)	0.065 seconds (2 buildings)
Concatenation	0.098 seconds	5.300 seconds
Radius	0.067 seconds	0.203 seconds
Building Height	0.091 seconds	Not Applicable

Table 6: Comparison of Fuseki Query Response Times between Helsinki and New York City Datasets.

7.3.5 SPARQL

Applying some basic SPARQL queries provided additional insight into both datasets. The Helsinki dataset contains 1,911,692 triples, while the New York City dataset includes 6,287,270 triples. Both datasets consist of city objects, specifically buildings, with the Helsinki dataset representing 677 buildings and the New York City dataset representing 23,777 buildings. In terms of geometry, the Helsinki dataset uses "Solid" geometry, whereas the New York City dataset utilizes "MultiSurface" geometry. Additionally, the Helsinki dataset features both LoD1 and LoD2, while the New York City dataset is represented at LoD2. These characteristics are summarized in Table 7.

Query	Helsinki	New York City
Number of Triples	1,911,692	6,287,270
Number of City Objects	677	23,777
Number of Buildings	677	23,777
Types of Geometry	Solid	MultiSurface
Types of LoDs	LoD1 and LoD2	LoD2

Table 7: Comparison of Basic SPARQL Query Characteristics between Helsinki and New York City Datasets.

7.3.6 GeoSPARQL

GeoSPARQL queries for the union, concatenation, and buffer operations were consistently applied across both datasets using the GeoSPARQL functions *union* and *buffer* to retrieve the results. However, the intersection operation required a different approach between the two datasets.

For the intersection queries, both the Helsinki and New York City datasets utilized the GeoSPARQL functions *sfIntersects* and *intersection*. The *sfIntersects* function was employed to determine whether a building intersected with a specified building, while the *intersection* function was used to obtain the Well-Known Text (WKT) representation of the intersection.

In the case of the New York City dataset, an additional step was necessary due to the complexity of its geometry. The conversion from 3D to 2D introduced self-intersecting polygons, which caused errors when using the *intersection* function. To address this issue, the boundary of the building was used instead of the complex geometry to search

for intersections and calculate the intersection WKT. This approach circumvented the problems associated with self-intersecting polygons and enabled accurate determination of intersecting buildings.

In contrast, the Helsinki dataset allowed for the retrieval and calculation of building heights due to the presence of a non-standardized building height attribute. This made it possible to calculate the average building height within the city. Unfortunately, this analysis could not be replicated for the New York City dataset due to the absence of a corresponding height attribute.

7.4   Case Study Summary

This chapter presented the application of the conversion tool, as well as the execution of SPARQL and GeoSPARQL queries[8] on two datasets: Helsinki and New York City. In the next chapter, we will discuss the challenges and limitations encountered during the course of this thesis.

---

[8] All the queries and commands used in this chapter are available at the following link: `https://github.com/aly1551995/CityJSON-LD/tree/main/Case%20Study`.

8 Discussion

This chapter showcases some challenges and limitations encountered throughout the thesis.

8.1 Challenges

Developing the conversion tool was a complex and tedious process involving several challenges.

8.1.1 Accurate Vocabulary

One of these challenges was formulating a vocabulary that accurately captured CityJSON's objects and their relations. The vocabulary needed to preserve semantics and promote the use of other vocabularies, such as the GeoSPARQL vocabulary used to define the WKT string of the 2D geometry of a CityJSON object.

8.1.2 Efficient Validation

Another significant challenge, besides accurate output, is efficiency. The tool required handling large files, reading them, validating them, converting them, and writing them back to disk in an acceptable amount of time. The validation step, in particular, was the bottleneck of the process, as it relied on a third-party library like PySHACL. Validation proved to be a significantly time-consuming process when handling large files, especially those with many vertices, as their recursive definition in our vocabulary proved computationally expensive to validate.

8.1.3 Formation of WKT

Finding the best way to generate the WKT string was difficult. The initial approach was to construct it manually, but then the Shapely package seemed more suitable. Besides generating the string, there was the obstacle of converting the given 3D geometric shapes to a 2D one without losing much information. Our initial approach was to project by removing the Z-coordinate and then compute the convex hull. However, the approach proved inaccurate compared to just projecting without any other functions applied to the geometry.

8.2 Limitations

In addition to challenges, the development process faced several limitations.

8.2.1 Non-Supported Features

While some optional CityJSON features such as appearances, geometry templates, extensions, and geometry semantics were not supported in this project due to their added complexity and time constraints, the core components of CityJSON were successfully converted. Our work focused on the essential elements of CityJSON, providing a robust foundation for the conversion process. By prioritizing these core features, we ensured that the most critical aspects of CityJSON were fully integrated into our vocabulary and the

conversion tool. This approach not only facilitated the successful conversion of the primary dataset elements but also laid the groundwork for future expansions. The structure we developed is extendable, offering a strong starting point for incorporating additional features and further enhancements.

8.2.2 Data Scarcity

The lack of CityJSON data, especially CityJSON data with specific geometry types such as *CompositeSurface*, *MultiSolid*, and *CompositeSolid*, rendered the testing process difficult. Nevertheless, the tool works well with the example CityJSON files found on their original website for these geometries. In addition, there was an attempt to create more CityJSON data by converting CityGML data to CityJSON using citygml-tools similar to the method discussed in 4.3. However, the method was cumbersome and out of the scope of this thesis. It only provided conversion to solids as geometry types; this type already had much data for testing purposes.

8.3 Discussion Summary

This discussion outlines the key challenges, such as creating an accurate vocabulary that encompasses the CityJSON semantics, efficient validation of the output and PySHACL bottleneck, and the approach to generating the WKT string for the geometry. The chapter also outlines the limitations faced during the duration of the thesis, including the time constraints and complexity for the non-included features and the hard-to-come-by CityJSON data.

9 Conclusion

In this chapter, we will summarize all the accomplishments throughout the thesis and reflect on future work to develop the tool further.

9.1 Summary

This thesis investigated the conversion of CityJSON to JSON-LD to enhance semantic interoperability and data integration for 3D urban models. We began by explaining the formats used for 3D urban data, such as CityGML and CityJSON, and their limitation, which, while effective in certain aspects, fall short of providing the semantic richness needed for integration with the Semantic Web. We then identified JSON-LD as a prominent RDF format for adding semantics to CityJSON.

We then showed the feasibility of converting CityJSON to JSON-LD by developing a conversion tool. We explained the approach and the implementations we took in developing the conversion tool, which included:

- A vocabulary that extended the original CityJSON semantics with a WKT string representation of the geometry object to allow for using GeoSPARQL. The leverage of GeoSPARQL allows powerful spatial functions and relations to be used.

- A SHACL file for validating the tool's output allows for avoiding the usage of OWL while still validating the output against the engineered vocabulary.

- A CLI tool with multiple arguments for flexible management of the conversion process.

With the tool's completion, we began showcasing its capability on two real-world datasets, Helsinki and New York, by converting them and uploading them to Apache Fuseki to apply queries using SPARQL and GeoSPARQL. The two use cases also allowed us to benchmark the tool's performance on two datasets with different characteristics.

We encountered and overcame many challenges during the tool's development, including creating the vocabulary, handling large datasets, and adding WKT. Despite these challenges, we developed the tool successfully, but there are still some limitations, such as deciding which features to support and the limited number of testing datasets. There is also potential for future improvements with better vocabulary, adding the missing features, and a GUI for a better user experience.

In conclusion, this thesis provides a proof of concept for more interoperable 3D city models. By bridging the gap between CityJSON and JSON-LD, the research paves the way for using CityJSON with the Semantic Web in urban modeling and planning via a conversion tool, which is the primary outcome of this thesis.

9.2 Future Work

Although the tool works as intended, there still could be room for several improvements to enhance the functionality and performance of the CityJSON to JSON-LD conversion tool.

9.2.1 Optimizing the Conversion Process

Optimize the conversion process to handle large datasets more efficiently. The optimization could involve applying a more efficient approach for processing and validating data, such as processing the CityJSON file in batches; the idea came from uploading the output of New York to Fuseki, which failed if the whole file was uploaded together, but when batch uploaded, it succeeded especially with the splitting of the vertices object on multiple uploads. Another option if the vertices are the source of the bottleneck is not to validate the vertices while validating the other object; this could allow mitigating the PySHACL recursion depth problem, especially when processing vertices to process the data faster.

9.2.2 Adding Missing Optional Features and Extensions

Expanding the tool's support to include optional CityJSON features and extensions would broaden its applicability. These missing elements are not part of the core CityJSON specification but are optional features and extensions that provide additional capabilities. For example, appearances enable more detailed visual representations, such as material and texture attributes, which are essential for realistic visualization of urban environments. Similarly, CityJSON extensions allow for the inclusion of specialized data and semantics tailored to specific applications. By incorporating support for these features, the tool could be used in a broader range of scenarios.

9.2.3 Vocabulary Enhancement

Enhancing the semantic expressiveness of the converted JSON-LD data is an area for future work. These enhancements involve developing more comprehensive ontologies and vocabularies that better capture the complexities of CityJSON and incorporate the missing CityJSON features. Such Ontologies and vocabularies enable more advanced queries and data integration possibilities.

9.2.4 Graphical User Interface (GUI)

Developing a GUI with interactive elements for the CLI tool will facilitate and improve the user experience. It also provides the potential addition of visualization and preview capabilities, which can be helpful for more insights and debugging in case of errors.

9.2.5 GIS Integration

Finally, future work should also focus on the tool's integration with existing Geographic Information Systems (GIS), such as QGIS, and its potential to facilitate interoperability and data sharing across different platforms.

References

[1] Gilles-Antoine Nys and Roland Billen. "From consistency to flexibility: A simplified database schema for the management of CityJSON 3D city models". In: *Transactions in GIS* 25.6 (2021), pp. 3048–3066 (page 3).

[2] Karwan Jacksi and Shakir M Abass. "Development history of the world wide web". In: *Int. J. Sci. Technol. Res* 8.9 (2019), pp. 75–79 (page 5).

[3] Jasmin Praful Bharadiya. "Artificial intelligence and the future of web 3.0: Opportunities and challenges ahead". In: *American Journal of Computer Science and Technology* 6.2 (2023), pp. 91–96 (page 5).

[4] Angus Stevenson. *Oxford dictionary of English*. Oxford University Press, USA, 2010 (page 5).

[5] Nicola Guarino, Daniel Oberle, and Steffen Staab. "What is an ontology?" In: *Handbook on ontologies* (2009), pp. 1–17 (page 5).

[6] Li Ding, Pranam Kolari, Zhongli Ding, and Sasikanth Avancha. "Using ontologies in the semantic web: A survey". In: *Ontologies: A Handbook of Principles, Concepts and Applications in Information Systems* (2007), pp. 79–113 (page 6).

[7] Antonio Hernández-Illera, Miguel A Martínez-Prieto, and Javier D Fernández. "Serializing RDF in compressed space". In: *2015 Data Compression Conference*. IEEE. 2015, pp. 363–372 (page 6).

[8] Tim Berners-Lee. *Linked Data - Design Issues*. 2006. URL: https://www.w3.org/DesignIssues/LinkedData.html (page 6).

[9] Florian Bauer and Martin Kaltenböck. "Linked open data: The essentials". In: *Edition mono/monochrom, Vienna* 710.21 (2011) (page 7).

[10] Richard Cyganiak. *The Linked Open Data Cloud Diagram*. http://id.loc.gov/vocabulary/iso639-2/eng. Date created: 2014-08-30 04:00:00.000. 2014. URL: http://lod-cloud.net/ (page 7).

[11] Bob DuCharme. *Learning SPARQL: querying and updating with SPARQL 1.1*. " O'Reilly Media, Inc.", 2013 (page 8).

[12] Yingjie Hu and Wenwen Li. "Spatial data infrastructures". In: *arXiv preprint arXiv:1707.03969* (2017) (page 9).

[13] M Caprioli, A Scognamiglio, G Strisciuglio, and E Tarantino. "Rules and standards for spatial data quality in GIS environments". In: *Proc. 21st Int. Cartographic Conf. Durban, South Africa 10–16 August 2003*. 2003 (page 9).

[14] Open Geospatial Consortium. *OGC CityGML Standard*. Accessed: 2024-06-28. 2023. URL: https://www.ogc.org/standards/citygml (page 9).

[15] CityJSON. *CityJSON: Now an OGC Standard*. Accessed: 2024-06-28. 2023. URL: https://www.cityjson.org/news/2023/11/06/cityjsonv2-ogc (page 9).

[16] Gerhard Gröger, Thomas H Kolbe, Claus Nagel, and Karl-Heinz Häfele. *OGC City Geography Markup Language (CityGML) Encoding Standard*. Tech. rep. 12-019. Open Geospatial Consortium, 2012 (pages 11, 12).

[17] Filip Biljecki, Hugo Ledoux, and Jantien Stoter. "An improved LOD specification for 3D building models". In: *Computers, environment and urban systems* 59 (2016), pp. 25–37 (page 12).

[18] Stelios Vitalis, Ken Arroyo Ohori, and Jantien Stoter. "CityJSON in QGIS: Development of an open-source plugin". In: *Transactions in GIS* 24.5 (2020), pp. 1147–1164 (page 12).

[19]   Hugo Ledoux, Ken Arroyo Ohori, Kavisha Kumar, Balázs Dukai, Anna Labet-ski, and Stelios Vitalis. "CityJSON: A compact and easy-to-use encoding of the CityGML data model". In: *Open Geospatial Data, Software and Standards* 4.1 (2019), pp. 1–12 (pages 12–14).

[20]   Markus Lanthaler and Christian Gütl. "On using JSON-LD to create evolvable RESTful services". In: *Proceedings of the third international workshop on RESTful design*. 2012, pp. 25–32 (page 15).

[21]   World Wide Web Consortium et al. *RDF 1.1 Primer*. Tech. rep. World Wide Web Consortium, 2014 (page 15).

[22]   Martin J Dürst. "Internationalized resource identifiers: From specification to test-ing". In: *19th International Unicode Conference*. 2001 (page 15).

[23]   Gregg Kellogg, Pierre-Antoine Champin, and Dave Longley. *JSON-LD 1.1 – A JSON-based Serialization for Linked Data*. Technical Report. W3C Working Draft. W3C, 2019. URL: https://hal.archives-ouvertes.fr/hal-02141614v1 (page 15).

[24]   Diego Vinasco-Alvarez, John Samuel Samuel, Sylvie Servigne, and Gilles Gesquière. "Towards a semantic web representation from a 3D geospatial urban data model". In: *SAGEO 2021, 16ème Conférence Internationale de la Géomatique, de l'Analyse Spatiale et des Sciences de l'Information Géographique*. hal-03240567. La Rochelle [Online Event], France, May 2021, pp. 227–238 (page 18).

[25]   Chih-Yuan Huang, Yao-Hsin Chiang, and Fuan Tsai. "An ontology integrating the open standards of city models and Internet of things for smart-city applications". In: *IEEE Internet of Things Journal* 9.20 (2022), pp. 20444–20457 (page 18).

[26]   Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. "SHACL: A de-scription logic in disguise". In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer. 2022, pp. 75–88 (page 23).

[27]   Open Geospatial Consortium. *Geographic information — Well-known text repre-sentation of coordinate reference systems*. Tech. rep. OGC Document Number: 18-010r11. Accessed: 2024-07-23. Open Geospatial Consortium, Mar. 2019. URL: https://docs.ogc.org/is/18-010r11/18-010r11.pdf (page 28).

[28]   Dongming Guo, Erling Onstein, and Angela Daniela La Rosa. "An improved ap-proach for effective describing geometric data in ifcOWL through WKT high order expressions". In: *An Improved Approach for Effective Describing Geometric Data in ifcOWL through WKT High Order Expressions* (2021) (page 28).

[29]   Vishal Jain and Mayank Singh. "Ontology development and query retrieval using protégé tool". In: *International Journal of Intelligent Systems and Applications* 9.9 (2013), pp. 67–75 (page 35).

[30]   Leon Martin and Andreas Henrich. "Specification and Validation of Quality Cri-teria for Git Repositories using RDF and SHACL." In: *LWDA*. 2022, pp. 124–135 (page 37).

[31]   W3C RDF Data Shapes Working Group. *Shapes Constraint Language (SHACL)*. Tech. rep. Accessed: 2024-07-15. World Wide Web Consortium (W3C), 2017. URL: https://www.w3.org/TR/shacl/ (page 38).