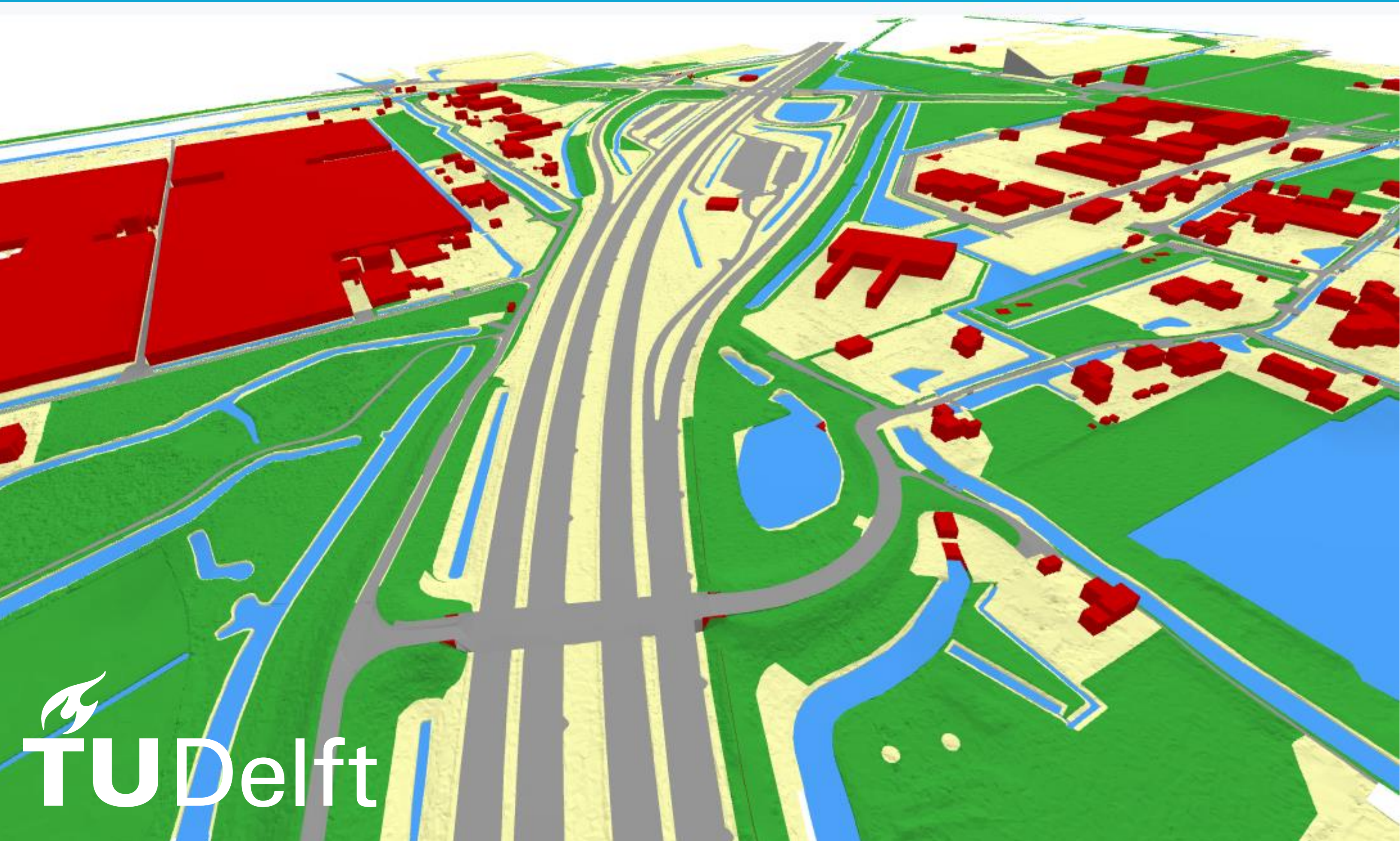
MSc thesis in Geomatics

CityREST: CityJSON in A Database + RESTful Access

Xiaoai Li

2021



TU Delft

MSc thesis in Geomatics

CityREST: CityJSON in A Database + RESTful Access

Xiaoai Li

June 2021

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors: Dr. Hugo Ledoux
ir. Jordi van Liempt
ir. Stelios Vitalis
Co-reader: ir. Balázs Dukai

Abstract

With the increasing demand for 3D city models in various applications, providing efficient access to 3D city models on the web has become very important and valuable. However, there is a lack of web services in which users can directly and effectively access the city objects contained in the 3D city models on the web. Another problem with web services related to 3D city models is that the size of 3D city models can become very large. As a consequence, the browser will not respond immediately until all city objects in the requested 3D city model have arrived. Additionally, although file-based systems are the easiest way of data storage, they have deficiencies in web services that require scalability, efficiency, and flexibility in data access. To fix these issues, the goal of this research is to develop the an efficient way of disseminating 3D city models on the web, to discover the method of enabling 3D city models to be parsed incrementally and efficiently by the browser, and to explore a proper database solution for storing 3D city models to support the fast data access.

CityJSON has advantages in web applications over the CityGML data format, thus being chosen as the main datasets used in this research. Inspired by *OGC API – Feature*, a RESTful API for fast access to geospatial features in CityJSON has been developed. The second part of my research is related to the data streaming. CityJSONFeature has been proposed to enable CityJSON to be parsed incrementally on the web. To improve the overall data streaming performance, I proposed two methods to stream the data from the database to the RESTful API so that the RESTful API can immediately start constructing the first CityJSONFeature and sending it to the user. The third part of my research is to explore a proper database solution for CityJSON datasets to best support the fast data access in the RESTful API. In addition, some auxiliary data is added into the database to improve the RESTful API's capabilities with efficient data filtering and streaming.

To evaluate the efficiency of the implemented methodology, a systematic benchmarking has been performed on three aspects: 1) the database schema, 2) the two streaming methods, and 3) the performance difference between the DBMS and the file system. The results show that in most use cases the DBMS can better support the RESTful API than the file system with the help of the built-in query and index mechanism. In addition, the overall streaming performance is largely improved when adding data streaming from the DBMS to the RESTful API. Lastly, based on the benchmarks, an optimal database schema has been chosen to support the RESTful API with fast data access, efficient data filtering and streaming.

Preface

The report in front of you is written to conclude my study at the Delft University of Technology for the degree of Master of Science in Geomatics. This thesis could not have been completed without the support of several people, who I would like to thank beforehand.

Firstly, I want to express my gratitude to Hugo Ledoux for this fascinating thesis topic and for his critical aiding mindset during the meetings that were shown to be crucial in the development of the overall quality and depth of this project. Thank you Jordi van Liempt for proving me with lots of useful guidance and well structured feedback. Next, a special thanks to Stelios Vitalis for his patience and kindness in helping me with the database schema design. I also want to show my gratitude to the external committee members Dirk Dubbeling and Balázs Dukai for taking time to attend my phased meetings.

In addition, I want to thank my friends for the great student time. A special thanks to my boyfriend who supported me during my master and lighted up my life whenever I felt demotivated. Lastly, I wish to thank my parents and my twin brother for their steadfast support since the beginning of this wonderful journey.

Xiaoai Li
Delft, June 2021

Acronyms

OGC	Open Geospatial Consortium	2
REST	Representational State Transfer	xiii
URL	Uniform Resource Locator	11
CRS	Coordinate Reference System	xix
API	Application Programming Interface	xiii
DBMS	Database Management System	xiv
JSON	JavaScript Object Notation	10
JSONB	JavaScript Object Notation Binary	xiv
WFS	Web Feature Service	11
WMS	Web Map Service	11
NoSQL	Not only Structured Query Language	14
SQL	Structured Query Language	16
NDJSON	Newline-delimited JSON	18
ROI	Region of Interest	xiv
LoD	Level of Detail	38
WKT	Well-known Text	xix
SFC	Space Filling Curve	19
HTML	HyperText Markup Language	46
WKB	Well-known binary	64
ADE	Application Domain Extension	7
RDBMS	Relational Database Management System	15
ADE	Application Domain Extensions	7
CQL	Common Query Language	13

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Nowadays, 3D city models are used for a broad range of applications and have been widely recognized as a potential to create social value. Biljecki et al. [2015] made a comprehensive overview on the state of art of 3D city modelling and identified more than 29 distinct use cases related to 3D city models (see Figure 1.1). However, compared to 2D spatial datasets, there are more difficulties in 3D dataset acquisition, reconstruction, maintenance and dissemination due to more complex structure and massive file sizes. Hence, the 3D city models are usually used at local scale and the application with 3D city models is thus limited.

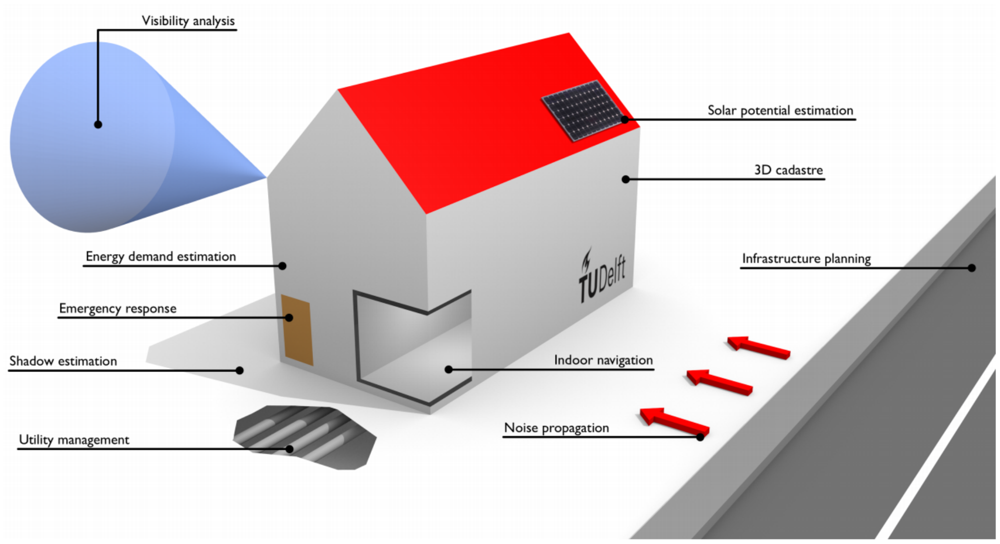
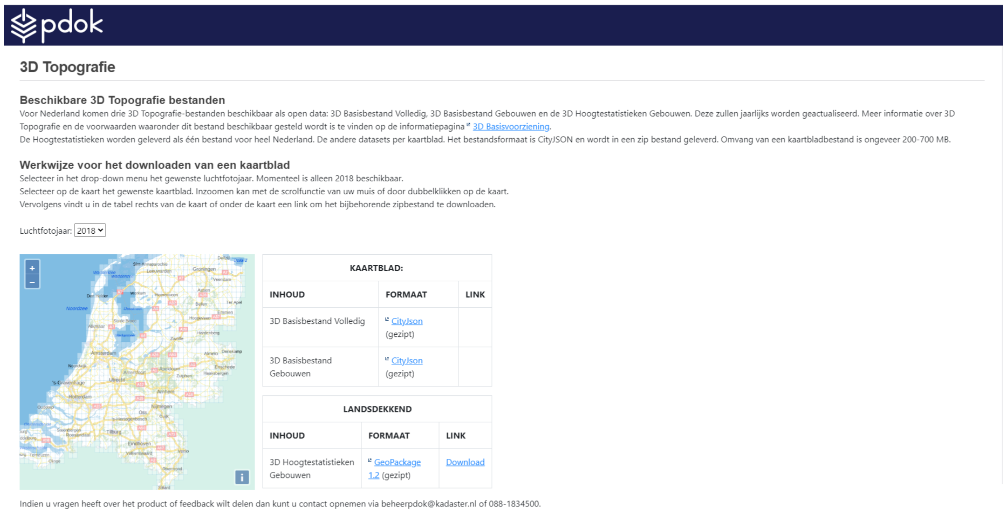


Figure 1.1: Applications in 3D city models (Figure from Biljecki et al. [2015]).

In the last decade, several scholars and organizations have been working on improving the dissemination and usage of 3D city models. In 2020, the Dutch Kadaster and the 3D Geoinformation research group at TU Delft collaborated to generate and disseminate 3D city models on the web [Dukai et al., 2020]. The workflow in the collaboration covers 3D data automated reconstruction from existing countrywide data (i.e. AHN and footprints) and enabling the 3D city models accessible through the national governmental geoportal, PDOK (see Figure 1.2).



3D Topografie

Beschikbare 3D Topografie bestanden
Voor Nederland komen drie 3D Topografie-bestanden beschikbaar als open data: 3D Basisbestand Volledig, 3D Basisbestand Gebouwen en de 3D Hoogtestatistieken Gebouwen. Deze zullen jaarlijks worden geactualiseerd. Meer informatie over 3D Topografie en de voorwaarden waaronder dit bestand beschikbaar gesteld wordt is te vinden op de informatiepagina 3D Basisvoorziening.
De Hoogtestatistieken worden geleverd als één bestand voor heel Nederland. De andere datasets per kaartblad. Het bestandsformaat is CityJSON en wordt in een zip bestand geleverd. Omvang van een kaartbladbestand is ongeveer 200-700 MB.

Werkwijze voor het downloaden van een kaartblad
Selecteer in het drop-down menu het gewenste luchtfotojaar. Momenteel is alleen 2018 beschikbaar.
Selecteer op de kaart het gewenste kaartblad. Inzoomen kan met de scrolfunctie van uw muis of door dubbelklikken op de kaart.
Vervolgens vindt u in de tabel rechts van de kaart of onder de kaart een link om het bijbehorende zipbestand te downloaden.

Luchtfotojaar: 2018

KAARTBLAD:		
INHOUD	FORMAAT	LINK
3D Basisbestand Volledig	CityJson (gezipt)	
3D Basisbestand Gebouwen	CityJson (gezipt)	

LANDSDEKKEND		
INHOUD	FORMAAT	LINK
3D Hoogtestatistieken Gebouwen	GeoPackage 1.2 (gezipt)	Download

Indien u vragen heeft over het product of feedback wilt delen dan kunt u contact opnemen via beheerpdok@kadaster.nl of 088-1834500.

Figure 1.2: The access to 3D city models through the Dutch geoportal, PDOK

However, the city objects contained in these 3D city models are not directly queryable through the geoportal, which means that users cannot directly browse them online and look into the city objects in detail (such as city object types). Instead, the entire file needs to be downloaded by the user and checked with other software. Moreover, these 3D city models are simply divided into fixed-size tiles and the size of one tile with various city objects can exceed 2 GB. Due to the lack of filtering function, the downloaded files with large sizes are difficult to process and use in applications. In addition, there is no 3D viewer provided for users to quickly inspect the city objects. Even if a 3D viewer is provided, the 3D visualization of a large tile cannot be performed very well in terms of the long response time. Because the browser will not immediately start the 3D visualization until all requested city objects have arrived. Suggested by W3C Working Group [2017], data formats should be modified or designed to enable the stream-based delivery when handling large amounts of geospatial data. In addition, although the file-based systems are the easiest way of data storage, they have deficiencies in some applications that require scalability, efficiency and flexibility in data access. For these reasons, the goal of this research is to develop the best way of disseminating 3D city models on the web so that users can quickly and efficiently access 3D city models, and to explore a proper database solution to support the fast data access.

In order to promote the interoperability and machine-readability of 3D city models, Open Geospatial Consortium (OGC) has been developing international standards for publishing and interoperating 3D city models in the last two decades [Van Den Brink et al., 2018]. CityGML was proposed by the OGC as an open data model and format for 3D city models. However, CityGML as an XML-based exchange format is quite difficult to processed, managed and disseminated [Pispidikis and Dimopoulou, 2018]. CityJSON, a JSON-based exchange format of 3D city models is less verbose and more web-friendly [Ledoux et al., 2019]. Hence, CityJSON is chosen as the main dataset used in this research to implement the efficient dissemination of 3D city models on the web. In 2019, OGC released a multi-part standard named *OGC API – Feature*. The core of this standard follows the RESTful architecture and specifies how geospatial resources are defined in the RESTful API [Open Geospatial Consortium, 2020]. The resource-oriented data access architecture can make the geo services more web-friendly for both users and developers.

Inspired by *OGC API – Feature*, I aim to develop a RESTful API for fast access to geospatial features in CityJSON. According to the some characteristics in CityJSON datasets and my research scope, there are some divergences between *OGC API – Feature* and the final implementation of the RESTful API. The second part of my research is related to the data streaming. CityJSONFeature has been proposed by Ledoux [2020] to enable CityJSON data to be parsed incrementally on the web. To improve the overall data streaming performance, I propose two methods to stream the data from the database to the RESTful API so that the RESTful API can start constructing the first CityJSONFeature and sending it to the user without having to wait for all data tuples to be transferred from the database. The third part of my research is to explore a proper database solution for CityJSON datasets to best support the fast data access in the RESTful API. In addition, some auxiliary data is also added into the database to improve the RESTful API's capabilities with efficient data filtering and streaming. In the end, I evaluated the design of the database schema, two different streaming methods and the performance difference between the DBMS and the file system regarding the efficient data access through the RESTful API.

1.2 Research challenges

3D city models consist of more complex structures and larger file sizes in comparison to 2D datasets. Hence, there are some challenges in CityJSON storage and dissemination that have to be solved. These challenges are formulated as followed:

- **Resource mapping**: Although *OGC API – Feature* defines the resources in the geospatial context, the specification of mapping the 3D data in CityJSON to those resources needs to be proposed. In CityJSON, one feature may be divided into several city objects, for example, the multi-part building. Thus, one geospatial feature may be associated with more than one city object. Correctly bundling the associated city objects together is necessary during data access.

- **Data filtering**: As mentioned in Section 1.1, a data filtering function is necessary for efficient data access and use. Regarding filtering, there are two aspects: semantic-based and geometric-based. Although the DBMS provides the indexing mechanisms to accelerate the filtering process, some auxiliary data is needed to support the filtering and there will be trad-off between the storage size and query performance.

- **Data streaming**: The requested data can be very large, especially when dealing with 3D datasets. The long response time is not user-friendly. In order to improve the web performance, modifications should be implemented to enable CityJSON data to be incrementally parsed by the browser. Considering that the CityJSON datasets will be hosted on the database, correctly streaming the data from the DBMS to the RESTful API is also necessary.

- **Data storage**: The conventional file system is the easiest way for data storage but it usually lacks scalability, especially when large amounts of data need to be managed. Besides, the data processing in a file-based system is less flexible than in a DBMS. Therefore, a DBMS is preferred for better data availability and access performance. The specific database schema for CityJSON datasets needs to be properly designed to support the fast data access in the RESTful API.

- **CRS issues**: The city objects in one CityJSON dataset share the same CRS, which is stored at the CityJSON object level. During the data access, the CRS information should be attached to geospatial features. However, effective data transmission should also be ensured, for example, during a streaming process. This means, the duplicate CRS information should be removed. Besides, different CityJSON datasets can have different CRSs and the geometric filtering process can also have different CRS issues. Therefore, proper CRS transformation is necessary.

1.3 Research objectives

The main research question for this thesis is:

- *How to best develop a RESTful API for fast access to geospatial features in CityJSON and how to properly store CityJSON in a database to support the fast data access?*

To answer this question, the following sub-questions should be considered:

- How to mapping the data in CityJSON datasets to the resources defined by *OGC API – Feature*?
- How to solve the CRS issues during data access?
- How to implement the efficient querying of city objects based on geometries and semantics?
- How to define streaming in the geospatial context and how to stream CityJSON datasets?

- Which database is suitable for storage of CityJSON datasets to support RESTful access and how to design the database schema?

1.4 Research scope

The research scope concerns the following aspects:

- **Datasets**: The datasets used in this research are CityJSON datasets. Considering the complexity, the property “extension”,“appearance”, “geometry-templates” in CityJSON objects are out of scope for this research.

- **Minimum accessible resource**: By following the *OGC API – Features*, I defined the minimum accessible resource as a city object. This means that the data access and filtering process only works at the city object level.

- **REST API**: This thesis focuses on the efficient access to CityJSON datasets on the web. Other resource operations such as modification and deletion are not investigated in this research.

- **Bounding box filtering**: Bounding box filtering only works on datasets with reference information. In addition, when performing bounding boxes on multiple CityJSON datasets, the filtered city objects should have the same CRS for maintaining the performance of the RESTful API.

- **Attribute filtering**: Unlike bounding box filtering, attribute filtering is only performed on a single CityJSON dataset. Moreover, to maintain high-performance attribute filtering in the RESTful API, attributes containing few distinct string or numeric values are only considered as queryables.

1.5 Research outline

The rest of the thesis is organized as follows:

- Chapter 2 discusses the related work, starting from the introduction of 3D city model to the general concepts of the RESTful API and the current database solutions for 3D city models.

- Chapter 3 introduces the methodology for this research, which covers the two main parts: developing a RESTful API and storing CityJSON data in a database.

- Chapter 4 covers the implementation details for prototyping the RESTful API.

- Chapter 5 presents the methodology used for the benchmarking. Three parts are concerned: database schema, streaming method and storage system. The benchmarks are analyzed to suggest the optimal solution for this research.
- Chapter 6 gives a conclusion of this research with the future work.

Chapter 2

Theoretical background and related work

2.1 3D city model

3D city models can be seen as digital models that represent 3D geometric entities (e.g. buildings, bridges and roads) of urban areas [Billen et al., 2014]. Compared to traditional 2D maps where city objects are represented in a symbolic way, 3D city models can display a real-realistic urban scenarios with texturing support [Wendel et al., 2017]. Hence, 3D city models can be beneficially used in urban visualization, analysis and applications. This can help urban planner and policy makers to get a better insight of the built environment.

3D city models have initially been used for simulating the built environment. Hence, the semantics and topology of city objects are largely ignored in the 3D city models. However, the lacking of semantic information limits the diversity of 3D city model related applications. Another issue is the interoperability of 3D city models. 3D city models from heterogeneous sources can not be directly used in the same application. To tackle these problems, OGC has defined an international standard for 3D city models and an XML-based data exchange format, named CityGML [Gröger et al., 2012].

2.1.1 CityGML

CityGML data model is developed to improve the interoperability by unifying the representation of the basic geometries and semantic information in a 3D city model [Gröger et al., 2012]. The geometry part stores geometrical and topological information in a consistent way. The semantic part complies to the ISO 19100 standards to enrich the geometry model for various use cases [Kolbe, 2009]. CityGML data model comprises 10 core semantic modules, which are extensible to other domains using the Application Domain Extensions (ADE) (see Figure 2.1).

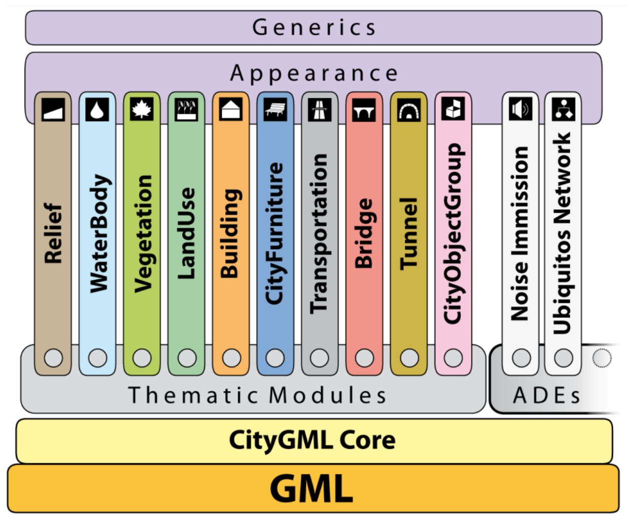


Figure 2.1: CityGML architecture (Source:virtualcitySYSTEMS).

2.1.2 CityJSON

CityJSON, a JSON-based encoding for the OGC CityGML 2.0.0 data model has been proposed by Ledoux et al. [2019]. A CityJSON file contains a CityJSON object of which the type is always "CityJSON" and the minimal valid CityJSON object must contain the properties shown in the Listing 2.1. The CityJSON object can also contain the member "metadata", where the reference system and the 3D geographical extent can be stored (see Listing 2.2). The member "vertice" stores a global list of the 3D vertex coordinates for the compression purpose, which is inspired by the Wavefront OBJ format.

```
{
  "type": "CityJSON",
  "version": "1.0",
  "CityObjects": {},
  "vertices": []
}
```

Listing 2.1: The minimal valid CityJSON object

```
{
  "metadata": {
    "referenceSystem": "urn:ogc:def:crs:EPSG::7415",
    "geographicalExtent": [ 84710.1, 446846.0, -5.3, 84757.1, 446944.0,
   40.9 ]
  }
}
```

Listing 2.2: Example of the property "metadata" at the first level of CityJSON object

A collection of city objects are stored in the property "CityObjects". The Listing 2.3 shows an example of one city object, which contains the property “type” whose value can be “Building”, “BuildingPart”, and other types of objects that are common in digital twins. The attributes for this city object are stored in the property “attributes”. The properties “parent” and “children” may exist when the city object is associated to the other city objects. The geometrical and semantic data for the city object are contained in the property “geometry”. From Listing 2.4 we can see one example of a geometry object, for which the type can be “MultiSurface”, “Solid” and other common 3D geometry type. The boundary presentation is used for storing the geometrical data and each vertex refers to the global vertex list.

```
{
  "id-1": {
    "type": "Building",
    "attributes": {
      "measuredHeight": 22.3,
      "roofType": "gable",
    },
    "children": ["id-2"],
    "geometry": [{...}]
  },
  ...
}
```

Listing 2.3: Example of One city object in CityJSON

```
{
    "type": "Solid",
    "lod": 2,
    "boundaries": [[
        [[0,3,2,1,22]],
        [[4,5,6,7]],
        [[0,1,5,4]],
        [[1,2,6,5]]
        ]],
    "semantics": {
        "surfaces" : [
        {"type": "RoofSurface" },
        {
         "type": "WallSurface",
         "paint": "blue"
        },
        {"type": "GroundSurface" }
        ],
        "values": [ [0, 1, 1, 2] ]
    }
}
```

Listing 2.4: Example of One geometry in one city object

To compress the CityJSON files, the 3D vertices can be transformed into integer values. The transformation matrix for obtaining the original coordinate values can be decomposed into the “scale” and “translate”, which are represented as arrays with 3 values and stored in the “transform” property (see Listing 2.5).

```
"transform": {
    "scale": [0.01, 0.01, 0.01],
    "translate": [4424648.79, 5482614.69, 310.19]
}
```

Listing 2.5: Example of one "transform" object in CityJSON

Compared to CityGML exchange format, CityJSON files are on average six times more compact proved by real-world datasets [Ledoux et al., 2019]. In addition to that, the JavaScript Object Notation (JSON) encoding is derived from JavaScript and is the preferred data exchange format for web applications [Freeman, 2019]. It is also natively supported by most programming languages, for example, Python. Due to the popularity of JSON data, some relational databases like *PostgreSQL* now support JSON data for storing and querying. Therefore, CityJSON has advantages in data storage and web applications, and has been chosen as the main datasets used in this research.

2.2 REST API

REST is acronym for Representational state transfer, which is a software architectural style that uses a subset of HTTP [T. Fielding, 2000]. There are 6 architectural constraints that have to be met in order to define an API as a RESTful API [Erl et al., 2012]. Those constrained are listed below::

- ***Uniform Interface***: This is an essential constraint to determine a RESTful API. It shows that no matter what type of device or application, there should be a unified way of interacting with the API.
- ***Stateless***: Statelessness means that the requests sent by the client can be understood by the RESTful API without the additional context data. The client must send all the necessary information to the RESTful API, such as query parameters to satisfy the stateless purposes.
- ***Cacheable***: The web browser can save some resources locally. When the browser sends the same request, there is no need to fetch the same resources from the server again. In this way, the number of client-server interactions can be reduced, thereby improving the scalability and performance.
- ***Client-Server***: The client-server architecture separates the resource providers and receivers, which can be developed independently.

- ***Layered System***: Between the client layer and the end server layer (resources), there can be intermediary layers. Those layers are grouped horizontally, which is similar to the IT communication structure. Each layer has specific function, for example, a security layer can be added to ensure the security policies in business [Richards, 2015]. The layered system helps develop new functions and makes it easy to manage the whole RESTful service.

- ***Code on Demand***: To meet the different demands on the client side, the RESTful API can optionally offer the code that can be run in the client side, for example, the Python-like expressions using Jinja or client-side scripts using JavaScript.

The key aspect in RESTful services is the resource [Jansen, 2011]. Resources can be grouped into collections and each collection itself is also a resource and contains one type of other resources in an unordered manner (see Figure 2.2). Each resource is identified by the Uniform Resource Locator (URL) and can be manipulated using HTTP verbs. The operations include create, read, update, and delete, which are implemented by POST, GET, PUT, and DELETE HTTP verbs, respectively.

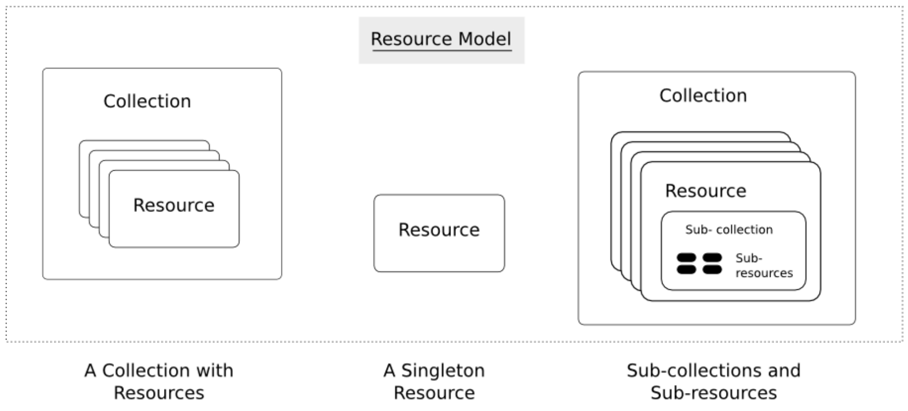


Figure 2.2: Resource model in REST API (Figure from Jansen [2011]).

2.3 OGC API - Features

As introduced in Chapter 1, the OGC released a multi-part standard named *OGC API – Features* in 2019. Compared to previous standards like Web Feature Service (WFS) 2.0 and Web Map Service (WMS), the core conceptual model of the *OGC API – Features* remained the same, enabling access to geospatial datasets through a web service [Holmes, 2017]. Prior to February 2019, the core part of the *OGC API – Features* was referred to as WFS 3.0, which uses a different approach compared to its predecessor WFS 2.0 in specifications. Although XML exchange format is still acceptable, JSON is prominently featured for services and the data encoding, along with HTML. Besides, the resource-oriented data access architecture (REST) is followed, which makes it more web-friendly for both users and developers. Therefore, it can be concluded that the *OGC API – Features* is web-oriented and consistent with the good practice on the web.

The *OGC API – Features* contains the following parts [Open Geospatial Consortium, 2020] and each part is discussed in details:

- ***Part 1**: Core standard*: The core of the *OGC API – Features* follows the RESTful architecture [Open Geospatial Consortium, 2019]. The resources defined by the *OGC API – Features* are listed in Table 2.1 and each resource provides links to the related resources[Open Geospatial Consortium, 2020].

Resource	**Path**	**Purpose**
Landing page	/	This is the top-level resource, which serves as an entry point.
Conformance declaration	/conformance	This resource presents information about the functionality that is implemented by the server.
API definition	/api	This resource provides metadata about the API itself.
Feature collections	/collections	This resource lists the feature collections that are offered through the API.
Feature collection	/collections/{collectionId}	This resource describes the feature collection identified in the path.
Features	/collections/{collectionId}/items	This resource presents the features that are contained in the collection.
Feature	/collections/{collectionId}/items/{featureId}	This resource presents the feature that is identified in the path

Table 2.1: The resources defined by OGC API (Table from Open Geospatial Consortium [2020])

  The *Feature collections* contains a list of key information about all the *Feature collection* resources. The key information must includes a unique identifier for the collection in the URL path and a list of CRSs of the geometries contained in the collection. Optionally, the description, title, spatial extent and temporal range of this collection can be included in the response. When a specific *Collection* resource is accessed, more details can be returned. The *Features* contains (part of) features in the identified collection. The response of *Features* also depends on the query parameters in the URL, for example, `bbox` and `datatime`. In addition, the URL parameter `limit` and `offset` can be added to support paged access to a large dataset without overloading the client. Each *Feature* can also be separately requested by its unique identifier.

- ***Part 2**: Coordinate Reference Systems by Reference*: One unavoidable issue when dealing with geospatial data is the CRS. The CRSs, on one hand, make geospatial data special and valuable to many real-life applications. On the other hand, different CRSs can cause difficulties in data management, filtering, integration and applications. In addition, the lack of CRS information is problematic since the coordinates for the geospatial data will be ambiguous, which may lead to failure in geometric and topological analysis.

  The first part of the *OGC API – Features* only supports one CRS: WGS 84. In order to address the CRS issue, the *Part 2: Coordinate Reference Systems by Reference* extends the first part's capabilities to use any other CRSs [Open Geospatial Consortium, 2020]. To avoid unnecessary duplication and ambiguousness of CRS identifiers, OGC provides a global list of supported CRS identifiers. The CRS identifiers can be used in the spatial extent (see Listing 2.6) and as a parameter in the URL (see Listing 2.7).

```
"extent": {
        "spatial": {
            "bbox": [
                [
                    -180,
                    -90,
                    180,
                    90
                ]
            ],
            "crs": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
        }
```

Listing 2.6: The spatial extent information

```
.../collections/buildings/items?crs=http://www.opengis.net/def/crs/
   EPSG/0/7415&...
```

Listing 2.7: Retrieving features from a collection in a supported CRS

- ***Part 3**: Filtering and Common Query Language (CQL)*: This part has not been officially published yet and only the draft version is available. It mainly works on enhancing filtering capabilities using the CQL in the API. The queryables that can be used to construct filter expressions for one collection is accessed by the URL: /collections/{collectionId}/queryables. Apart from the queryables, a comprehensive set of operators are also defined by OGC, which includes logical operators, comparison operators, spatial operators and temporal operators. In the CQL, a key/value is used to express a filtering statement where the key refers to a operator and the value contains the operands. An example of the CQL for a BETWEEN operation is given in Listing 2.8.

```
{
  "between": {
     "value": { "property": "depth" },
     "lower": 100.0,
     "upper": 150.0
  }
}
```

Listing 2.8: An example of a BETWEEN predicate

- ***Part 4**: Simple Transactions*: This part is still in a draft stage. It adds an extension to the first part by supporting other resource operations like modification and deletion.

The *OGC API – Features* provides a good guidance and inspiration in the development of a RESTful API for fast access to geospatial features in CityJSON. The first part offers a template for mapping CityJSON datasets to the resources in the RESTful API at geospatial context and the second part gives ideas for solving the different CRS issue. Part 3 helps with the implementation of data filtering for efficient data access and use while the last part is out of my research scope since the data access is the only resource operation considered in the RESTful API for CityJSON.

However, the draft document of Part 3 is not available until most parts of my methodology is well designed and implemented. Therefore, some of my methodology and decisions are not fully in line with Part 3. Regarding Part 1, there is no need to return a list of CRSs of the geometries contained in one *Feature Collection* because all features in one CityJSON dataset share the same CRS. In addition, the only filterable resource specified in Part 1 is the *Features*, which means that the filtering process only operates on a single CityJSON dataset. When the region of interest defined by the user covers multiple CityJSON datasets, the RESTful API can not perform this spatial filtering task. Although Part 3 extends the filtering capability by allowing cross-collection queries, the approach proposed in Part 3 requires the pre-selected set of collections as a URL parameter (see Listing 2.9). Part 2 can handle the CRS issues by providing each CRS with a unique URL. Another identifier can be used to simplify the expression of CRSs and the CRS parsing in the RESTful API, for example, the EPSG code. The main divergences between *OGC API – Features* and my implementation will be concluded and discussed in Section 6.1.1.

```
http://www.someserver.com/ogcapi/search?collections=collection1,
   collection3&filter=prop1=10 AND prop2>45
```

Listing 2.9: Multi-collection filter example

2.4 Database management system

The relational database is proposed by Codd [1970]. It uses a logical structure to allow users to easily access specific data that is related to one another. In a relational database, data is organized into tables where each row is a record with a unique ID and each column is assigned a specific datatype such as an integer number. With the changing requirements, the evolution of the internet and the larger variety of data types that occurred from this evolution, relational databases struggles particularly with the scalability and speed demanded by users in the mid-1990s [D. Foote, 2018]. This led to the development of Not only Structured Query Language (NoSQL) databases [NoSQL, 2021]. NoSQL databases use different data structures (i.e. key-value, document) and avoid the same constraints as traditional relational databases [Drake, 2019]. NoSQL databases are mostly distributed databases, which means that they scale horizontally by adding more nodes in a cluster environment when the data size increases [Olivera, 2019].

Despite the more scalability and flexibility of NoSQL databases, relational databases continue to have their place in many applications. Especially, when the reliability and consistency in the data transmission are required. Before a relational database can be populated and used, a strict and thorough requirement analysis is needed to understand the real-world relations between the entities so that the database can function correctly [Polding, 2018].

2.4.1 JSON in DBMS

The JSON files can fit naturally into NoSQL systems such as MongoDB. However, storing JSON data in Relational Database Management System (RDBMS)s is challenging because there is no traditional data type and operations that are suitable for the JSON data. Despite their flexibility, NoSQL systems have some major disadvantages compared to traditional RDBMSs [Chasseur et al., 2013]. There is no standardized query language in the NoSQL systems, which hinders the query capabilities and efficiency. Hence, NoSQL systems usually suffer from handling complex data processing, such as filtering by multiple requirements. ACID (atomicity, consistency, isolation, durability) properties in the RDBMSs help maintain the consistency during transactions [Haerder and Reuter, 1983]. However, this is not guaranteed in NoSQL systems.

Therefore, the integration of JSON data into a RDBMS has been explored and discussed by many scholars and organizations. Chasseur et al. [2013] proposed a mapping layer named Argo that allows the RDBMS to support JSON data with flexibility. With the mapping layer, ACID properties are guaranteed and complex operations on JSON data are supported, such as join and aggregation in the RDBMS. Liu et al. [2016] presented an automatically calculated dynamic soft schema for the JSON data, which narrowed the gap between the traditional RDBMSs and the schemaless NoSQL DBMSs. A query-friendly and self-contained binary format for encoding JSON has also been proposed to shrink the performance difference between the traditional data types and schema-free JSON type.

ISO released a new international SQL standard (ISO/IEC 9075:2016) in 2016 [ISO, 2016]. This new standard supports JSON data by storing the JSON as a string and defining the related functions to interpret the string to a JSON. Some major DBMS vendors have integrated JSON into their database schema so that the database users can freely store, query, and index JSON data. Oracle follows the new SQL/JSON standard [ModernSQL, 2017] while PostgreSQL creates two new data types, namely JSON and JSONB [PostgreSQL, 2021b]. There are essential discrepancies between JSON and JSONB. In terms of storage, JSON data type stores an exact copy of the text, which is similar to the SQL/JSON standard, while JSONB is decomposed and stored using a binary format. Regarding performance, JSON is processed much slower due to the additional reparse. In addition, there are more operators and indexing mechanisms for JSONB in PostgreSQL. It can be concluded that JSONB has more values in applications

2.4.2 Database for 3D city models

Despite the fact that file-based systems are the easiest way of data storage, they have deficiencies in some applications that require scalability, efficiency and flexibility in data access. The

DBMS has become a popular way of data storage and has been able to provide an efficient solution for processing large amounts of data using Structured Query Language (SQL).

A 3D geo-database solution called *"3D City Database (3DCityDB)"* was developed to store CityGML datasets by using a spatially-extended RDBMS [3DCityDB Team, 2019]. CityGML is an object-oriented data model which defines a conceptual schema for the most relevant urban entities and specifies the relationship between them [Yao et al., 2018]. Therefore, the database solution for the CityGML data model can be abstracted to solving the problem of mapping the object-oriented data model onto a relational data model [Stadler et al., 2009]. There are strong reasons why a spatially-extended RDBMS is adopted in *3DCityDB*. Most spatial relational databases like *PostgreSQL* support spatial data types (geometry and geography) and functions, which offers an efficient way of spatial data storage, query and analysis. Besides, the spatial indexing mechanism helps to retrieve data faster, thereby improving data performance. In addition, the RDBMS ensures the integrity and consistency in data transactions, which are crucial for some applications, for example, for a cadastral application. NoSQL database solutions for 3D city models are also investigated by some researchers in recent years. Mao [2014] proposes a 3D city model management system with a NoSQL database, *Mongodb*, where 3D city models in different data formats such as Collada and X3D, are maintained with *CityTree*.

Since CityJSON follows the CityGML 2.0.0 data model, the data in CityJSON files can be theoretically stored into 3DCityDB. However, there are some drawbacks when using 3DCityDB for storing CityJSON:

- **Complex**: The schema group of 3DCityDB is huge and each city object type has several related tables, which increases the complexity (see Figure 2.3). Therefore, setting up the database schema of 3DCityDB requires complicated work, and it takes a long time to get familiar with the database schema. Those two characteristics show that 3DCityDB is not friendly for beginners to use.

- **Less flexible**: To extend the CityGML data model to meet new requirements, ADE has been proposed by Open Geospatial Consortium [2012]. The main purpose of ADE is to provide flexibility in dealing with different use cases with newly-defined classes and attributes [Biljecki et al., 2018]. However, in 3DCityDB, all the generic classes of city objects are mapped into specific tables and the corresponding generic attributes are mapped into the columns. This means that the ADE cannot directly work with 3DCityDB. New database schema for the ADE needs to be designed.

- **XML-oriented**: 3DCityDB is developed to store CityGML files and the data format the 3DCityDB handles is XML. This is an essential reason why the database schema in 3DCityDB is complex and verbose. As discussed in Section 2.4.1, some major DBMS vendors have supported JSON storage with flexibility. This feature has not been integrated into 3DCityDB, and it can be seen as a waste for CityJSON datasets.

- **Out of the research scope**: 3DCityDB has well-structured and fine-defined tables, like *SURFACE_GEOMETRY* and *THEMATIC_SURFACE*. However, this research focuses on the data access at city object level. Hence, it is not necessary to split a city object to several sub tables. In addition, it takes longer time to construct the original city objects from 3DCityDB, which leads to a performance penalty in the RESTful API.

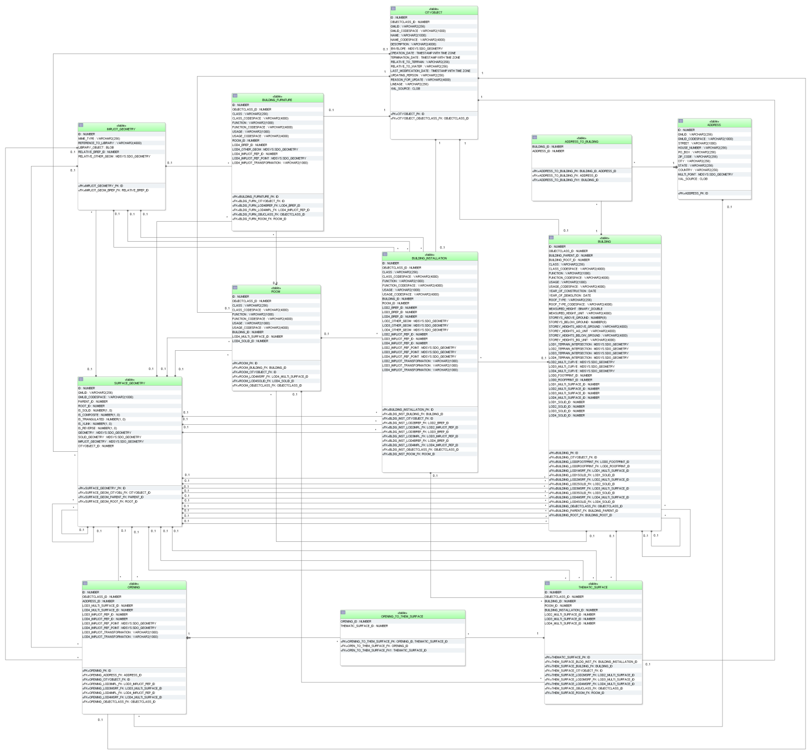


Figure 2.3: Building database schema in 3DCityDB (Figure from 3DCityDB Team [2019]).

Based on the above discussion, a better database solution is recommended to store CityJSON instead of directly adopting 3DCityDB. Staring [2020] explores and compares the performance of using NoSQL and SQL databases for CityJSON, and concludes that both database solutions can work well on most use cases. However, there are some limitations in the NoSQL database when handling geospatial data. Compared to the spatially-extended relational database, for example, *PostgreSQL*, there are not as many spatial functions provided in the NoSQL databases. Moreover, the CRS is currently limited to the WGS84 datum in *MongoDB*, which causes troubles in handling different CRS issues. In addition, the spatial indexing mechanism in *MongoDB* only supports 2D geometry types, so the data query performance for CityJSON is limited.

A more compact database schema is more efficient for data query and process, thereby facilitating better performance [Stadler et al., 2009]. In order to avoid a massive number of tables in the relational database schema, the JSONB data type introduced in Section 2.4.1 is used in Staring's schema to reduce the number of tables (see Figure 2.4).

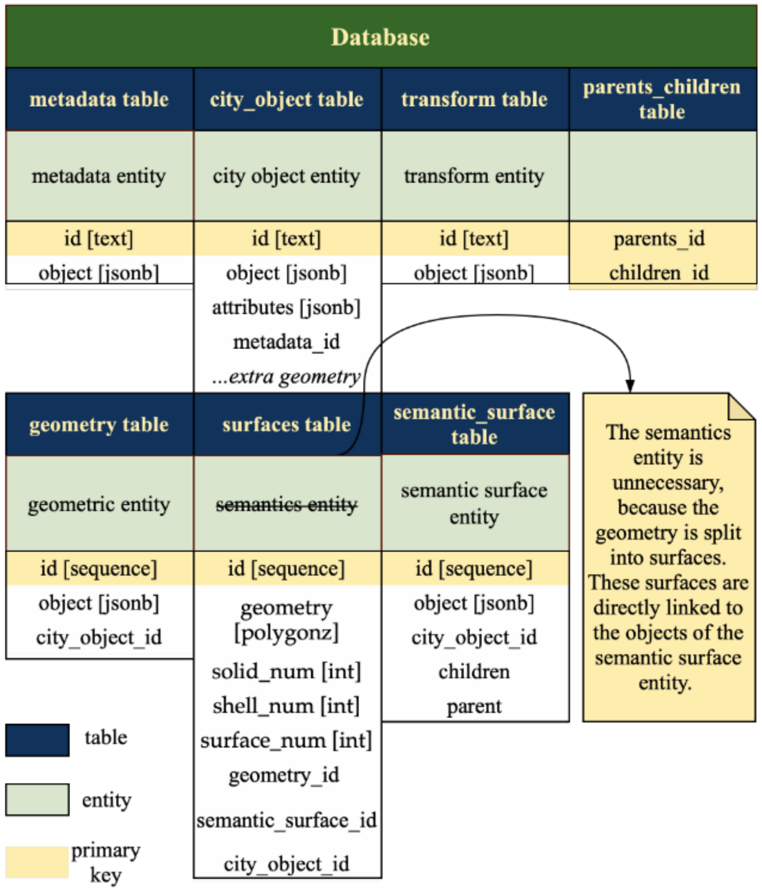


Figure 2.4: An overview of the relational database schema for CityJSON from [Staring, 2020].

In respect to the above, the relational database solution for the storage of CityJSON is first investigated in this research. Note that a file-based system can be more efficient than a DBMS in certain situations, for example, accessing a whole file. Therefore, a hybrid storage management integrating the file-based system and DBMS could be considered.

2.5 Data streaming

The W3C working group has documented some suggestions concerning the publication of spatial data on the web [W3C Working Group, 2017]. Regarding the data transmission, it is advised that data formats should be modified or designed to enable the stream-based delivery when handling large amounts of geospatial data.

Newline-delimited JSON (NDJSON) is a data format for JSON streaming [Long, 2021]. As the name implies, the newline character can be used as a delimiter to distinguish each valid JSON object, which can be parsed and processed independently without having to read the whole file into memory [Lück, 2018]. In order to enable GeoJSON data to be parsed incrementally on the web, GeoJSON has been modified to GeoJSON Text Sequence by being integrated with NDJSON [S. Gillies, 2017]. Therefore, the client application can start parsing and processing the first received feature before the whole data transmission is completed.

CityJSON cannot work directly with NDJSON due to the global list of vertices as mentioned in Section 2.1.2. For reconstructing the geometries for one city object, the entire file needs to be read into memory since the geometries are represented by having references to that global list of vertices [Ledoux et al., 2019]. In order to enable the streaming of CityJSON, CityJSONFeature has been proposed by Ledoux [2020], whose structure is as Listing 2.10. One CityJSONFeature contains a city object and its possible associated city objects. The indexing vertex mechanism is still kept with a local list of vertices. Each CityJSONFeature object is independent and delimited by a newline character, thus being streamable.

```
{
  "type": "CityJSONFeature",
  "id": "myid", //to identify which CityObject is the "main" one
  "CityObjects": {},
  "vertices": [],
  "appearance": {},
}
```

Listing 2.10: The structure of a CityJSONFeature

When the CityJSON data is hosted in the DBMS, another issue that can lead to high latency during data transmission is when the DBMS processes and returns all data tuples to the RESTful API at once. In this case, the RESTful API needs to wait until all the data tuples are transmitted, and then start constructing CityJSONFeature. Hence, it may take very long time for the clients to receive the first constructed CityJSONFeature. Therefore, how to properly stream the data from the DBMS to the RESTful API to improve the overall streaming performance will be discussed in Section 3.2.5.

2.6 Space filling curves

In the late 19th century, mathematicians have been interested in Space Filling Curve (SFC) [Lawder and King, 2000]. The Peano curve is the first SFC that has been created by Peano [1890] (see Figure 2.5). After one year, the Hilbert curve is proposed by Hilbert [1891], which is a variant of the Peano curve (see Figure 2.6). Another SFC called Morton curve (see Figure 2.7) is described by Morton [1966], who first applied it to sorting files. Those SFCs can be used to map 2-dimensional or multi-dimensional space to 1-dimensional space while the spatial coherence is preserved [Orenstein and Manola, 1987]. From the figures we can see that the main difference between these SFCs lies in the order in which they traverse multi-dimensional points [Zhou et al., 2021]. The traversal order can significantly influence the extent of spatial coherence. Compared to the Peano curve, the Hilbert and Morton curves have less significant "jumps" between points and can better preserve the proximity of points in multi-dimensional space.

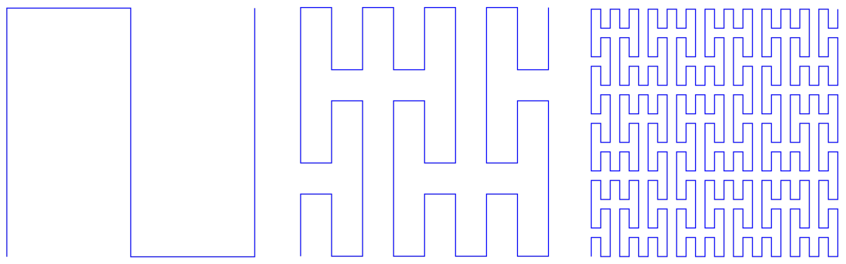


Figure 2.5: Three iterations of a Peano curve construction (Source: Wikipedia).

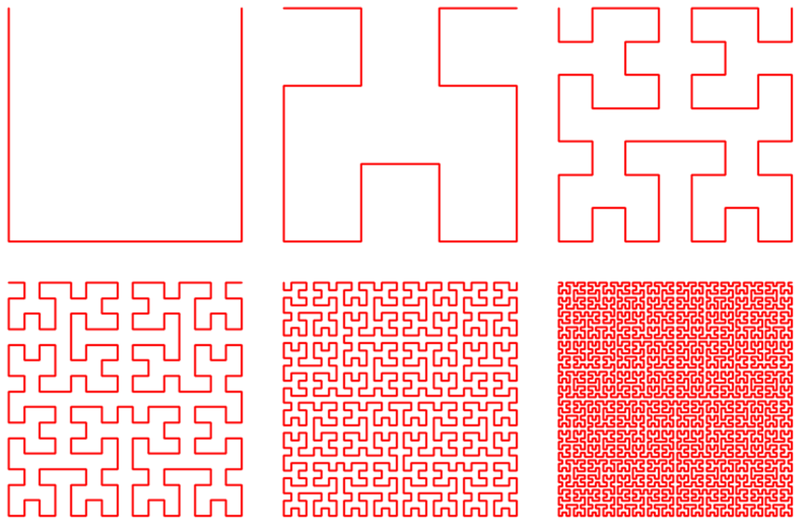


Figure 2.6: Six iterations of a Hilbert curve construction (Source: Stack Overflow).

Therefore, a Hilbert or Morton SFC can be used to index the center points of the city objects and indicate the spatial proximity between them. The SFC index can be used during data streaming process by sorting the query results using the SFC index. In this method, an extra column storing the index value is needed in the database schema. An alternative is to insert these city objects into the database by the SFC index order. In the second method, there is no extra ordering work during data query and no additional data column in the database. Another benefit from the second method is that these city objects will be clustered in the storage medium. Usually, nearby city objects are retrieved at the same time. Hence, consecutive physical blocks in the disk will be accessed instead of random ones [Psomadaki, 2016]. Since fewer physical blocks will be visited, the query performance can be improved. The detailed implementation of using a SFC to index city objects in CityJSON datasets is described in Section 4.2.2.

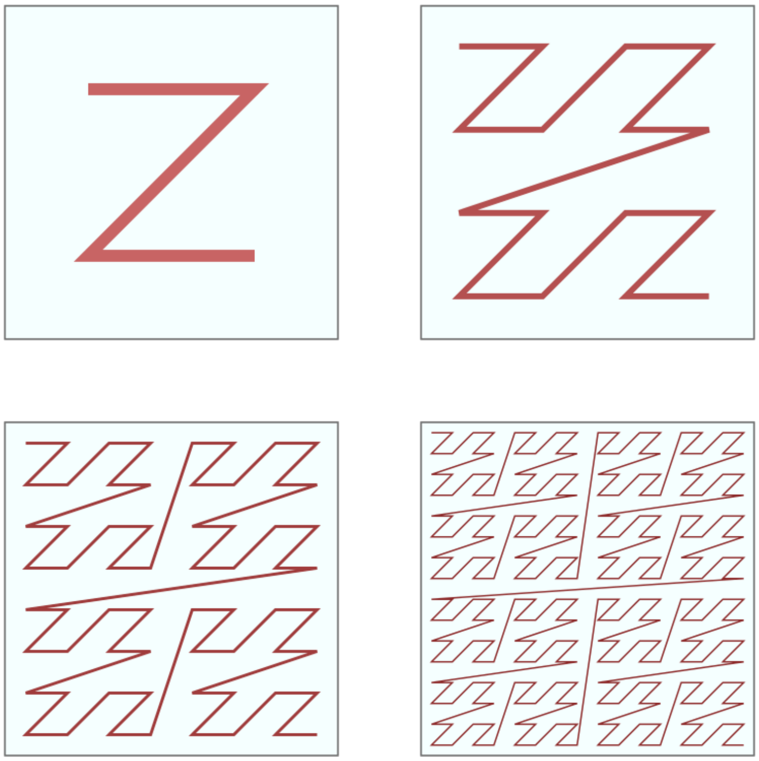


Figure 2.7: Four iterations of a Morton curve construction (Source: Wikipedia).

Chapter 3

Methodology

3.1 Overview

A brief overview of the methodology I have developed is depicted in Figure 3.1 and consists of three main parts:

- **Developing and implementing a RESTful API**: The API is built to allow access to geospatial features in CityJSON. The data in CityJSON is mapped to the resources defined by the *OGC API – Features* as mentioned in Section 2.3. API pagination mechanism allows the fast data browsing in a paginated way. The filtering function is developed for efficient data access and use. Data streaming can be used when handling large filtered data.

- **Storing CityJSON in a database**: A database schema is designed for CityJSON. Then the data in CityJSON files is parsed and during CityJSON parse, auxiliary data (e.g. bounding box & metadata of attributes) is generated to better support the API. All the data is inserted into the database.

- **Benchmarking**: Three aspects are benchmarked: database schema, streaming method, and storage system. All of them contribute to the web performance of the RESTful API. A systematic benchmarking for these aspects will be discussed in Chapter 5.

3.2 Developing a RESTful API

As mentioned in Section 2.3, developing a RESTful API allows great flexibility of data access. In this way the geofeatures are not tied to other resources and can be accessed in a RESTful way. Although most parts follow the OGC API – Feature, Some divergences between *OGC API – Feature* and the my methodology of the RESTful API are made according to the some characteristics in CityJSON datasets and my research scope.

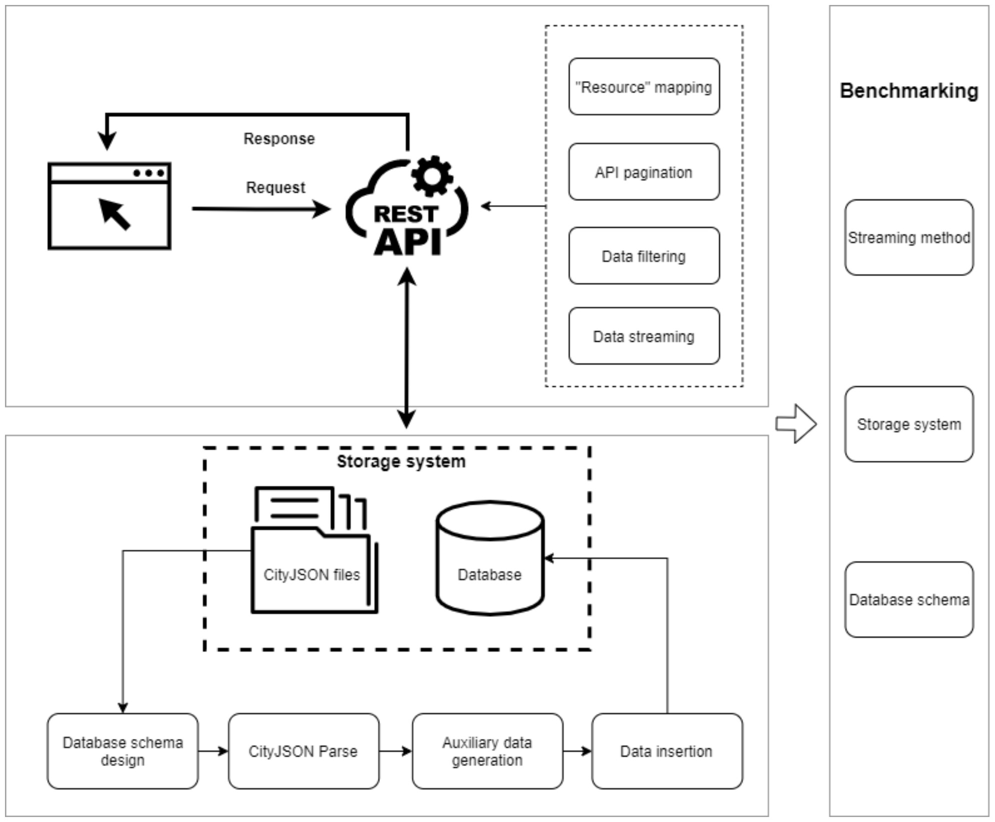


Figure 3.1: The overview of the methodology.

3.2.1 Resource mapping

The first step of developing a RESTful API is to define the resources. Inspired by the OGC, the resources at geospatial context can be classified into four groups: *Feature collections*, *Feature collection*, *Features*, *Feature*. The "resource" mapping of CityJSON datasets is proposed and the corresponding resource is listed in Table 3.1:

Resource	Path	CityJSON
Feature collections	/collections	An overview of all CityJSON objects
Feature collection	/collections/{collectionId}	An overview of one CityJSON object
Features	/collections/{collectionId}/items	One CityJSON object containing all city objects
Feature	/collections/{collectionId}/items/{featureId}	One CityJSONFeature

Table 3.1: The “Resource” mapping of CityJSON datasets.

- **Feature collections**: This resource provides an overview of all the CityJSON objects that are offered in this API which contains the name, description, CRS and 2D bounding box in WGS 84, attribute information for each CityJSON dataset.
- **Feature collection**: This resource presents an overview of the CityJSON object identified

in this URL. The overview for this CityJSON dataset is corresponding to one in the Feature collections

- **Features**: This resource lists all city objects contained in one CityJSON dataset and responses as a CityJSON object.
- **Feature**: This resource presents one city object identified in this URL. Note that the returned resource is a CityJSONFeature, where the top level city object is identified by the property id in the CityJSONFeature.

CityJSON data format is object-oriented, which makes it convenient for "resource" mapping. The value of the property "CityObjects" contains all the city objects with unique ids, which can be directly used for routing the resource *Feature*. However, only one city item may be split into several city objects in CityJSON (see Figure 3.2). Hence, when a city object at first level are associated with other city objects at second level, extra process is needed when returning the city object to the users.

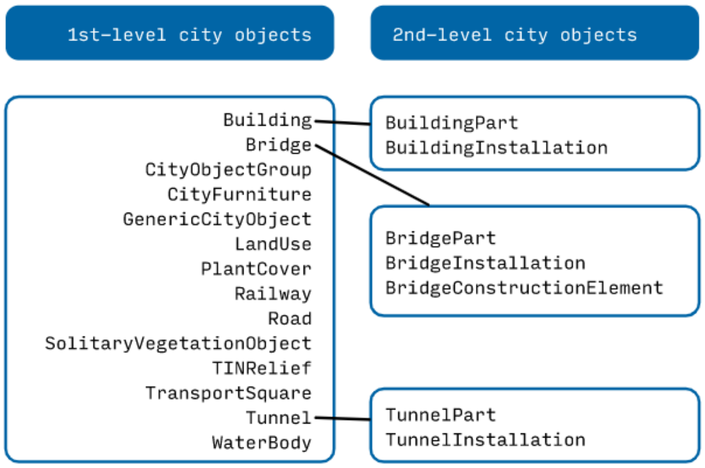


Figure 3.2: The types of city objects in the CityGML data model (Figure from [Ledoux et al., 2019])

Figure 3.3 shows an example of the multi-part building containing 2 building parts. The building is represented as 3 city objects in CityJSON (see Listing 3.1). The hierarchical relation is stored in the property "children" and "parents" and linked by the city object IDs. Therefore, when the building feature is accessed, its building parts should also be returned for the semantic reason. Additionally, from Listing 3.1 we can see that the geometrical information for the multi-part building is stored at the building part level. Hence, building parts should be included in the API response to reconstruct the multi-part building from a geometric perspective.

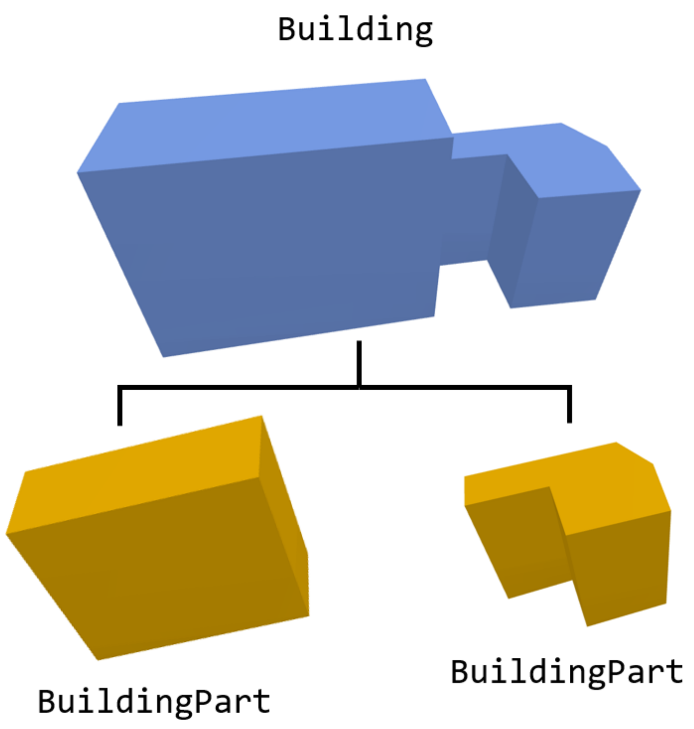


Figure 3.3: An example of the multi-part building.

```
"CityObjects": {
    "GUID_8CE54418-E2F7-49A7-9A8D-C3D172BA62C4": {
        "type": "Building",
        "geometry": [],
        "children": [
            "GUID_8CE54418-E2F7-49A7-9A8D-C3D172BA62C4_1",
            "GUID_8CE54418-E2F7-49A7-9A8D-C3D172BA62C4_2"
        ]
    },
    "GUID_8CE54418-E2F7-49A7-9A8D-C3D172BA62C4_1": {
        "type": "BuildingPart",
        "geometry": [...],
        "parents":  ["GUID_8CE54418-E2F7-49A7-9A8D-C3D172BA62C4"]
    },
    "GUID_8CE54418-E2F7-49A7-9A8D-C3D172BA62C4_2": {
        "type": "BuildingPart",
        "geometry": [...],
        "parents":  ["GUID_8CE54418-E2F7-49A7-9A8D-C3D172BA62C4"]
    }
}
```

Listing 3.1: An example of the multi-part biulding in CityJSON

3.2.2 API Pagination

When the user accesses the *Features* resource in a large CityJSON dataset, the API could return millions of city objects, resulting in high network traffic and browser crashing. A pagination mechanism helps in reducing the response size to make sure the API can run in a stable and effective manner.

To implement the API pagination, two parameters `limit` and `offset` are added into the path (see in Table 3.2). The default access follows the insertion order of city objects in a database.

Resource	Path	Response
Features	/collections/{collectionId}/items?limit={n} &offset={n}	A (sub)CityJSON object

Table 3.2: The access to *Features* in a paginated way

Although there are other methods to implement the pagination mechanism in a more robust way, the offset pagination is adopted in this research due to its simplicity. Additionally, the API works with a SQL database where `limit` and `offset` are already included in the SQL SELECT Syntax. The frequently discussed issue of offset paging is the inconsistency when new city objects are simultaneously inserted by users. But this will not be a problem in this API because the data access is the only resource operation considered in the RESTful API for CityJSON. Thus, the inconsistency problem of the offset pagination in this API is avoid.

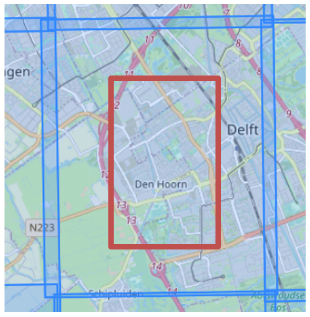
3.2.3 Bounding box filtering

Bounding box filtering is necessary for efficient data access and use. Users can select a ROI through a bounding box, and then download the filtered 3D city data for spatial processing. A bounding box can be defined by the two corner points and represented as an array with 4 values: `[MINX, MINY, MAXX, MAXY]`. Usually, when handling geographical data, the CRS is essential to connect the data to locations in the real world. Thus, a CRS should be bound to the bounding box as a second URL parameter.

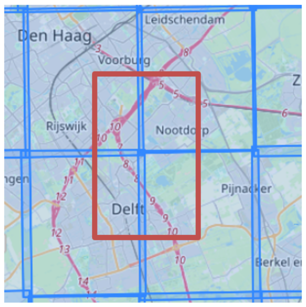
The ROI selected by the user can either be inside one CityJSON dataset or intersected with multiple CityJSON datasets (see Figure 4.9). For this thesis, two different URLs for accessing the filtered data have been designed.

- **Filtering one CityJSON dataset**: The two parameters `bbox` and `crs` are appended to the URL for accessing the filtered *Features* in one specific CityJSON dataset (see Table 3.3).

  When the RESTful API receives the request, the name of the CityJSON dataset, the bounding box and the CRS are parsed from the URL to build the SQLs. To construct a CityJSON object with complete information, the database will be queried twice. The first query is to retrieve some information of the to-be-filtered CityJSON dataset, which applies to all



(a) Filtering one CityJSON dataset



(b) Filtering multiple CityJSON datasets

Figure 3.4: Two cases for the bounding box filtering.

Resource	**Path**
Features	/collections/{collectionId}/items?bbox={[minx, miny, maxx, maxy]} &epsg= {n}

Table 3.3: *Features* filtered by a 2D bounding box inside one CityJSON dataset

the city objects, for example, the CRS and the transform matrix for compression. The second query is to implement the bounding box filtering. Note that the CRSs of the bounding box and the CityJSON dataset may be different, so CRS conversion is required in this case. Generally, CRS converting on the bounding box or on the geometries in CityJSON are both applicable. However, considering the processing efficiency, it is better to apply CRS conversion on the bounding box.

After the DBMS sends the filtered data tuple to the API, each city object is reconstructed and combined into a new CityJSON dataset. To reduce the response size, unused and duplicate vertices are deleted. Then the CRS and the "transform" obtained from the first query are added into the new CityJSON object. Also, the 3D bounding box of the new CityJSON object is calculated based on its vertex list and stored as metadata in "geographicalExtent". In the end, the new CityJSON object is sent to the user.

- **Filtering multiple CityJSON datasets**: When implementing the filtering on multiple CityJSON datasets, no specific CityJSON object is identified in the URL. Therefore, the two parameters `bbox` and `crs` are appended to *Collections* (see Table 3.4).

Resource	**Path**
Features	/collections/?bbox={[minx, miny, maxx, maxy]} &epsg= {n}

Table 3.4: *Features* filtered by a 2D bounding box intersected with multiple CityJSON datasets

The first step is to determine CityJSON datasets which are intersected with the bounding box. Then, the city objects within those CityJSON datasets are further filtered by the bounding box. After that, the filtered city objects from different CityJSON datasets need to be merged into one CityJSON and sent to the user.

Note that CityJSON datasets intersected with the bounding box may have different CRSs. As a result, city objects from different CityJSON datasets can not be directly merged into one CityJSON object. Although CRS conversion can solve this problem, the entire process of vertex CRS conversion is time-consuming and inefficient, which contradicts the original goal of this research: fast data access. Thus, the bounding box filtering on multiple CityJSON datasets is only performed when the filtered city objects share the same CRS.

Different from the first use case, the "transform" information cannot be simply inherited from one CityJSON dataset. Therefore, a new "transform" information needs to be generated.

3.2.4 Attribute filtering

In addition to geometric data, rich semantic data is stored in CityJSON datasets. A CityJSON dataset may contain various city objects in an area (such as buildings, bridges and water bodies). However, for example, when a user only needs city objects with the building type, city objects with other types become redundant, which can affect data access performance and cause difficulties in data processing. Therefore, attribute filtering function is very valuable in the RESTful API.

To obtain a better attribute filtering function through the API, the queryable attributes are sent to users in *Feature collection* along with other information of the CityJSON dataset. Not all attributes in CityJSON dataset can be queried since the query performance for some attributes can be very low. This is the case for an attribute with object values. To maintain high-performance attribute filtering in the RESTful API, attribute with string and numeric types are only considered as queryables.

In addition to sending the attribute names, the possible values are also included in the response. Depending on the value type and the number of different values, there are three scenarios for one attribute:

- **Numeric type**: An important feature of numeric values is that they can be well analyzed by statistics and summarized into a small number of meaningful values. Instead of listing each different value in the attribute, the maximum and minimum values are representatively sent to the user.

- **String type with few different values**: When an attribute has few different values, the city objects containing this attribute can be well categorized into a small number of groups, which helps to filter the city objects effectively.

- **String type with many different values**: The extreme case is that every city object has a different value for this attribute. Apparently, the response size can be very large when returning the attribute information. More essentially, these attributes are less useful and not worthy of filtering, thus not considered as queryables.

Apart from the queryables, predicates in attribute filtering are also important. Table 3.5 shows a list of different operations. However, not all operations can be applied to every attribute. For example, comparison operators only work on attributes with numeric values while membership operator *in* can only be applied to attributes with string values. Note that when the enumeration only contains one value, the operator *in* implicitly performs the same with the operator = for the string-valued attributes. Logical operator *and* is used to connect multiple attributes and make sure the filtered results satisfy all the conditions.

Class	Operator	Abbr	Description
Comparison	=	eq	Equal to
	>	gt	Greater than
	<	lt	Less than
	>=	gt	Greater than or equal to e
	<=	lte	Less than or equal to
	range		Is within a range
Membership	in		Is present in an enumeration
Logical	and		All statements are true

Table 3.5: The different operations in attribute filtering

The next step is to construct the request URL for accessing *Features* filtered by attributes. From Table 3.6 we can see that the parameter `attrs` is appended to the path `/collections/collectionId/items`.

Resource	Path
Features	/collections/{collectionId}/items/?attrs={{}}

Table 3.6: The access to *Features* filtered by attributes

To facilitate the RESTful API to parse and process the request, the parameter value for `attrs` should be clarifiable and simple. Based on the two requirements, a JSON encoded string is chosen for the value. First, it is easy for the API to parse a JSON encoded string to a JSON object. In addition, multiple URL parameters can be avoided when combining different attribute filtering conditions (i.e. the logical operation) by storing every attribute filtering condition as a key-value pair. The key is the attribute name and the value stores the filtering condition. Moreover, due to the flexibility of key-value stores, different expressions of filtering conditions can be used. For attributes with string values, an array is used to store the enumeration values selected by the user. For attributes with numeric values, an object is used to explicitly store three properties: *operator, operand and range* and their corresponding values. Listing 3.2 shows an example of the `attrs` value.

```
{
    "type": [
        "Building"
    ],
    "roofType": [
        "1000",
        "1120"
    ],
    "AbsoluteEavesHeight": {
        "operand": 14.43,
        "operator": "lt",
        "range": [2,20]
    }
}
```

Listing 3.2: An example of the value of `attrs`

After receiving the request, the RESTful API first parses the JSON encoded string and converts it into an object, and then performs a loop operation on the object to construct the SQL with whole attribute filtering conditions. The rest of the steps are similar to the bounding box filtering section.

3.2.5 Data streaming

Sending a large volume of filtered data at once causes long periods of latency, which is undesirable for some applications (e.g. fast visualization). Instead, the data should be chunked into smaller ones and sent to the user in a streamed way. As mentioned in Section 2.5, CityJSON datasets cannot be parsed incrementally on the web due to the global list of vertices. To overcome this problem, CityJSON is decomposed into city objects as a series of CityJSONFeatures, which are then stored in a NDJSON where each CityJSONFeature object is independent and delimited by a newline character.

From Figure 3.5 we can see how the data is transmitted through the RESTful API. After the DBMS finishes data filtering, data tuples are sent to the API. Instead of constructing a large CityJSON object, the RESTful API asynchronously generates and sends a set of CityJSONFeatures to the user.

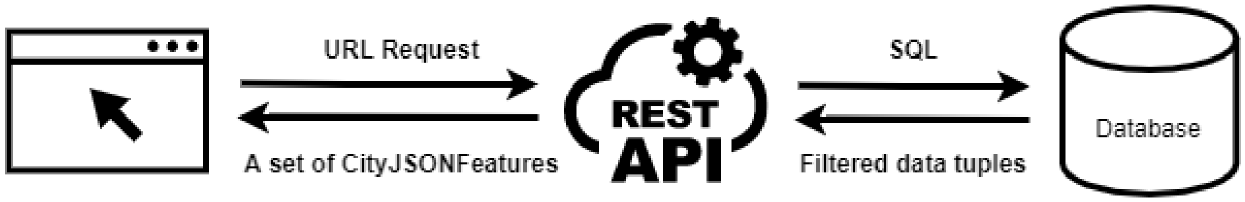


Figure 3.5: The data transmission in the RESTful API.

Some important information is stored at CityJSON level (such as the CRS and the transform matrix for integerization) and applies to all city objects. However, repeatedly storing these data in each CityJSONFeature will increase the response size and reduce the web performance. To fix this issue, a metadata object is generated and sent to the user before streaming city objects. The metadata object contains the important information and has a "type" property with the value of "MetadataCityJSONFeature", which can be used to differentiate the metadata object from CityJSONFeatures (see Listing 3.3).

```
{"type":"MetadataCityJSONFeature","metadata":{"referenceSystem":"..."},"
   transform":{}}
{"type":"CityJSONFeature","id":"id_0","CityObjects":{},"vertices":[]}
{"type":"CityJSONFeature","id":"id_1","CityObjects":{},"vertices":[]}
{"type":"CityJSONFeature","id":"id_2","CityObjects":{},"vertices":[]}
{"type":"CityJSONFeature","id":"id_3.","CityObjects":{},"vertices":[]}
```

Listing 3.3: An example of the streamed data in http response

Another issue that can lead to high latency and Internet overload is when the DBMS processes and return all filtered data tuples to the RESTful API at once. To solve this issue I proposed two methods for chunking the data from DBMS to the RESTful API in this thesis:

- **Paginating the query**: This method works similar to the API pagination as described in Section 3.2.2. The difference is that in API pagination the user needs to sent multiple requests to the RESTful API to browser all *Features* in one CityJSON dataset while in the data filtering case the RESTful API only receives one request, then paginates the query and sends requests to the DBMS multiple times (see Figure 3.6).

- **Streaming the query results**: The main difference from the first method is that the RESTful API only accesses the database once and all query results are generated on the database side (see Figure 3.7). Apparently, the query results should be chunked into smaller ones and streamed to the RESTful API.

Theoretically speaking, both methods can reduce the throughout and avoid network congestion to a certain extent during the data transmission from the DBMS to the RESTful API. However, both methods have advantages and disadvantages. When the query process is complicated, using the paging method, the RESTful API can receive the initial query result faster than

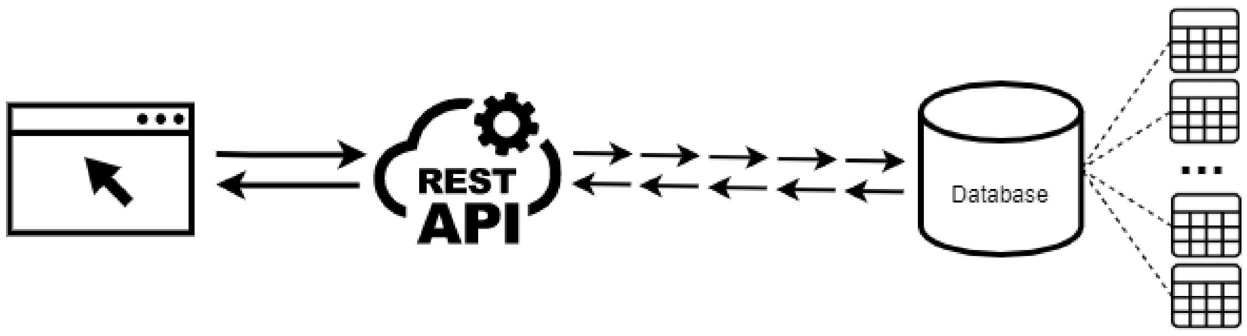


Figure 3.6: The workflow of paginating the query.

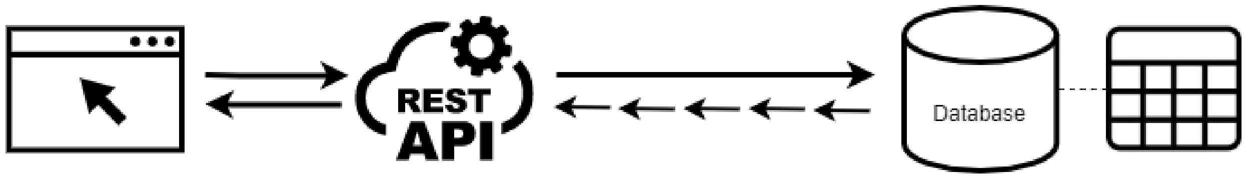


Figure 3.7: The workflow of streaming the query results.

using the second method, thereby providing the faster first response to the user. However, when the time for the DBMS to retrieve all the query results is insignificant, the second method is preferable because the database query is a time-consuming process. Multiple queries in the first method will cause performance degradation and it will take longer time for a user to receive a complete response. Anyhow, a systematic benchmarking for these two methods will be described in Chapter 5.

3.3 Storing CityJSON data in a database

Considering that the spatially-extended relational databases are still far superior to NoSQL databases when handling spatial datasets as concluded in Section 2.4.2, storing CityJSON data in a relational database has been investigated in this thesis. As mentioned in Section 2.4, a relational database schema for CityJSON is proposed by Staring [2020]. From Figure 2.4 we can see that city objects are fragmented into 3D surfaces, which are stored as 3D geometry type. Storing surfaces separately can be helpful for certain researches and applications where the smallest geometric unit is a surface (e.g. building energy modelling). However, the fragmentation is not necessary in this thesis since the minimal resource in the RESTful API is a city object. Additionally, it is difficult and time-consuming to reconstruct the city object from 3D surfaces. The 3D coordinates in the surfaces need to be extracted and then assembled according to the original hierarchical order. After that, these vertices will be converted to index representation and integerized. The entire process is complicated and will reduce the performance of the RESTful API.

To better support fast data access and streaming in the RESTful API, I propose a new database schema and the following aspects are considered during the database schema design:

- **Access to resources**: The database should support the RESTful and fast access to the resources listed in Table 3.1. Note that CityJSON is based on JSON, which can not fit in a traditional data type. To flexibly store CityJSON datasets in a relational database, special data types should be considered.
- **Parent-child relation**: The database should provide efficient query of associated city objects to help construct a complete *Feature*, in which a city object at the first level and its (possible) associated city objects at the second level are assembled together. This is very important for data stream processing, because a series of CityJSONFeatures needs to be generated quickly and correctly.
- **Bounding box filtering**: To achieve effective bounding box filtering in a database, auxiliary data can be added in the schema. Additionally, an appropriate index can be applied to the data to better support fast filtering.
- **Attribute filtering**: The semantic information contained in different city objects can be largely varied. Therefore, storing semantics in the database should have great flexibility, and can also effectively support attribute filtering. Additionally, an overview of attribute information for each CityJSON datasets should be generated and stored in the database to support the RESTful API.

3.3.1 Access to resources

The CityJSON object contains multiple properties: type, version, metadata, Cityobjects, vertices and transform. As can be seen from Figure 3.8, when accessing *Feature collections* and *Feature collections*, version and metadata are needed for the response. In the case of accessing *Features* and *Feature*, Cityobjects, vertices and transform along with version and metadata are required. Each *Feature collection* is identified by a CityJSON ID which is set to the CityJSON file name, and each *Feature* is differentiated by city object ID which is the key name in property Cityobjects.

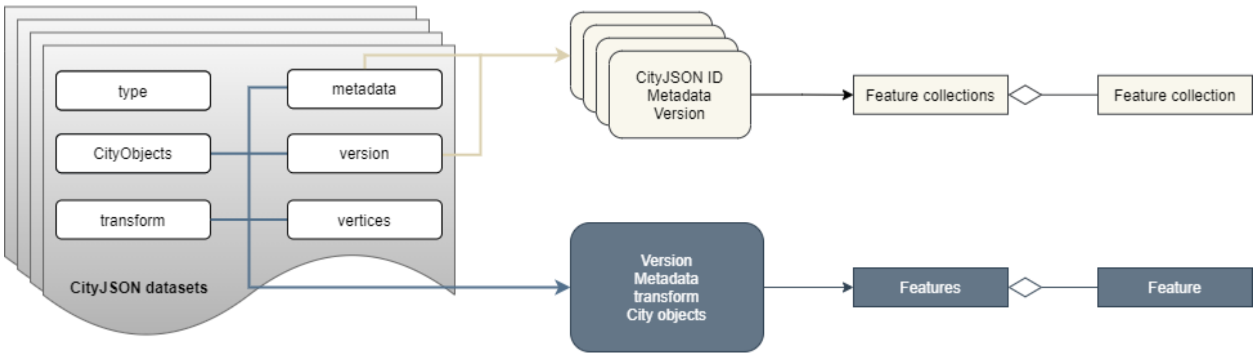


Figure 3.8: The mapping of data in CityJSON to resources in the RESTful API.

Based on the above analysis above, my proposal is to use two tables to store the CityJSON datasets. In this respect, one table named *Cityobject* that will be used for city objects, and the other table named *CityJSON* for other properties except the Cityobjects in CityJSON.

The next step is to determine how to store those properties in the tables. Most databases have a rich set of data types, such as `serial`, `int`, `varchar`. However, some properties are JSON objects and cannot be well supported by traditional data types. A text datatype is able to store the data presentation in the JSON object, but it lacks key/value querying and indexing capabilities. To overcome this problem, a JSONB datatype can be used, which is inspired by Staring [2020]. JSONB provides great flexibility in schemaless data storage. In addition, it is a binary storage format which helps reduce database size and data transmission time from the DBMS to the RESTful API. More essentially, JSONB supports the select operation on the single key/value pair and offers multiple options to index the JSON data.

When the database can support CityJSON storage, another issue that needs to be discussed is access efficiency. To avoid internet overload, API pagination mechanism is applied when accessing *Features*. However, the large list of vertices stored in the *CityJSON* table contradicts the pagination mechanism and may reduce the paging performance. This is because most irrelevant vertices will be queried from the database and sent to the RESTful API. Instead, the global indexing mechanism can be divided into local indexes at the city object level. To achieve this, a column called *vertices* is added into the *Cityobject* table. In addition, the vertex indices in the boundary presentation for each city object has been updated to the local list of vertices.

According to the above discussion, the database schema is initially designed and given in Figure 3.9.

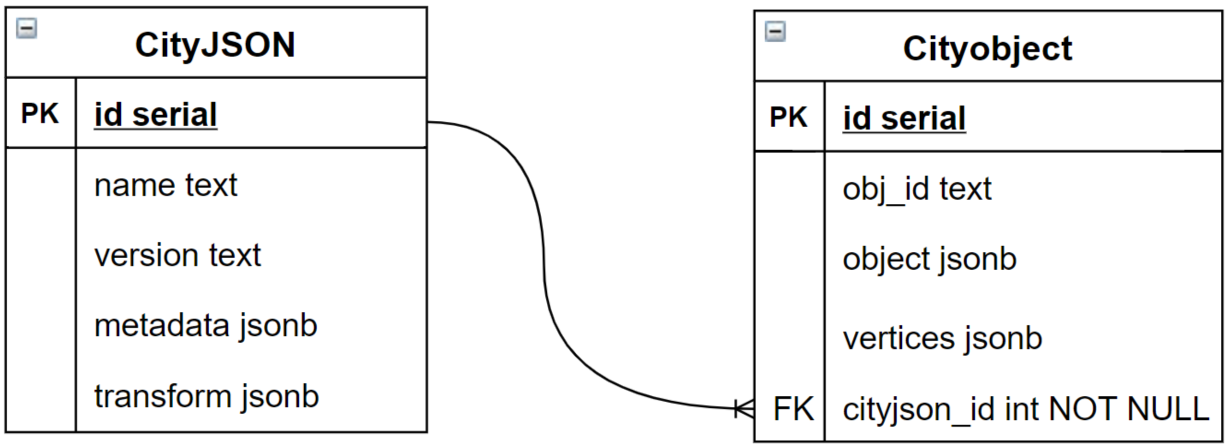


Figure 3.9: The database schema for supporting access to resources listed in Table 3.1.

3.3.2 Parent-child relation

The data contained in CityJSON objects is mapped to an entity-relational model (see Figure 3.10). CityJSON and city objects are the two entities, which are associated by the belonging relation. The entities are mapped to two tables in the database schema and the belonging relation is reserved by a foreign key in *Cityobject* table (see Figure 3.9).

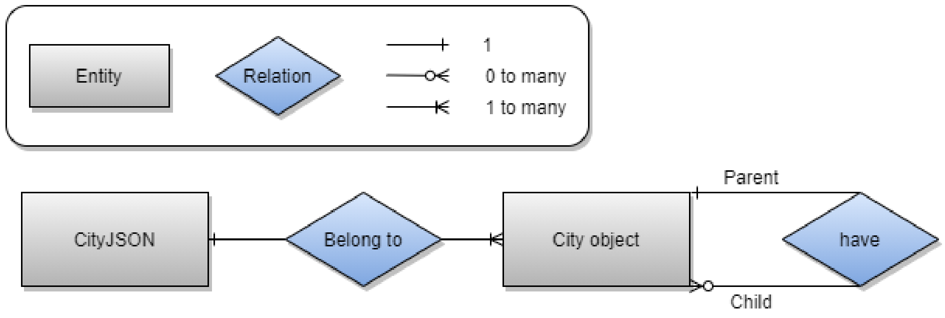
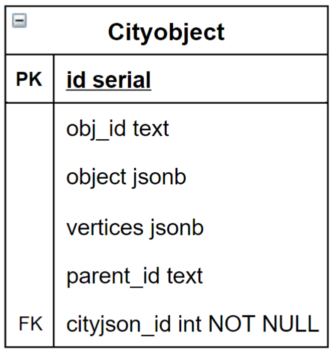


Figure 3.10: The defined entity-relational model.

However, the way to map the hierarchy relation between city objects to the database schema is not solved yet. To enable efficient query of associated city objects, three solutions have been proposed in this thesis :

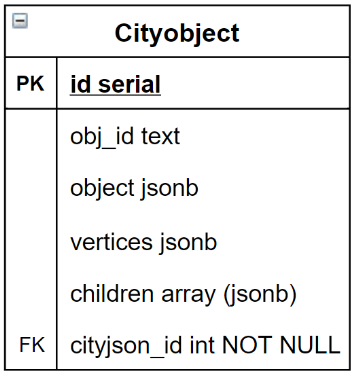
- **Adding a *parent_id* foreign column**: The traditional way to map the associations in entity-relational model to database schema is to add a foreign key or an associative table. Although the parent and children properties in city object are represented as two arrays, which indicates a many-to-many relation, they are one-to-many relation in reality. Thus, adding a *parent_id* foreign column with *text* type in the *Cityobject* table is sufficient to map the hierarchy relation (see Figure 3.11).



	Cityobject
PK	**id serial**
	obj_id text
	object jsonb
	vertices jsonb
	parent_id text
FK	cityjson_id int NOT NULL

Figure 3.11: The *Cityobject* table with a *parent_id* column.

- **Adding an extra column**: Considering that a more compact database schema may help improve the querying efficiency [Stadler et al., 2009], a *children* column can be added in *Cityobject* table to replace the *parent_child* table. The value of children data is represented as a text array, which is supported by the array type, `text[]` in the database. Therefore, it becomes easier to query the children's IDs. Another option is to store the children as JSONB type. When the array operation is required, JSONB can be cast to array type first. Figure 3.12 shows the modified *Cityobject* table with a *children* column



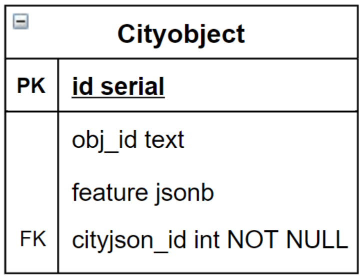
	Cityobject
PK	**id serial**
	obj_id text
	object jsonb
	vertices jsonb
	children array (jsonb)
FK	cityjson_id int NOT NULL

Figure 3.12: The *Cityobject* table with a *children* column.

However, obtaining children's IDs is not sufficient in most use cases. Instead, the complete child objects should be retrieved by querying *Cityobject* table with a list of children's IDs. One arising issue from the array type is that the query result of the children's IDs is a list of arrays in which the `IN` operator in the `WHERE` clause can not work. To solve this problem, `unnest(anyarray, ...)` function has been used to flatten multiple arrays to a set of values.

- **Storing complete *Features***: The query of associated city objects is mainly to help construct complete *Features*, which is of great importance in data streaming. Hierarchical relationships can be implicitly stored in *Features*, so that additional columns or tables are not needed.

  From Figure 3.13 we can see that a *feature* column is added to *Cityobject* table while *object* and *vertices* columns are removed to avoid redundancy at the same row.



	Cityobject
PK	**id serial**
	obj_id text
	feature jsonb
FK	cityjson_id int NOT NULL

Figure 3.13: The *Cityobject* table with a *feature* column.

In theory, storing complete *Feature*s in database can reduce the response time, thereby improving the web performance of the RESTful API. The (possible) associated city objects are stored in *Feature*s, so there is no need to query child city objects. Moreover, when adopting the first two solutions, an additional process of clustering city objects by the *parent_id* to generate correct *Feature*s is required, for which, `ORDER BY main_id` is appended to the end of SQL.

However, the third solution may lead to a larger database size compared to the first two. It is because that *Feature*s for city objects at the second level are also stored, which is redundant to the corresponding city objects at the first level. However, there is always trade-off between storage size and performance. When performance is stressed, it should be less strict on avoiding redundancy. Anyhow, the storage size and query performance will be measured, and a systematic benchmarking will be described in Chapter 5.

3.3.3 Bounding box filtering

The geometric information in city objects is complex. One city object may contain multiple geometry objects with different types and Level of Detail (LoD)s. To simplify the process of bounding box filtering, the spatial operation will be performed on the bounding box of the filter and the bounding boxes of the city objects. Therefore, the 2D bounding boxes of the city objects can be calculated in advance and stored in *Cityobject* table using an auxiliary column named *bbox*. Note that when filtering multiple CityJSON datasets, the first step is to determine which CityJSON datasets are intersected with the bounding box of the filter. Hence, a *bbox* column will also be added into *CityJSON* table. The database schema with bounding box data is depicted in Figure 3.14, where the data type for the bounding boxes are not determined yet.

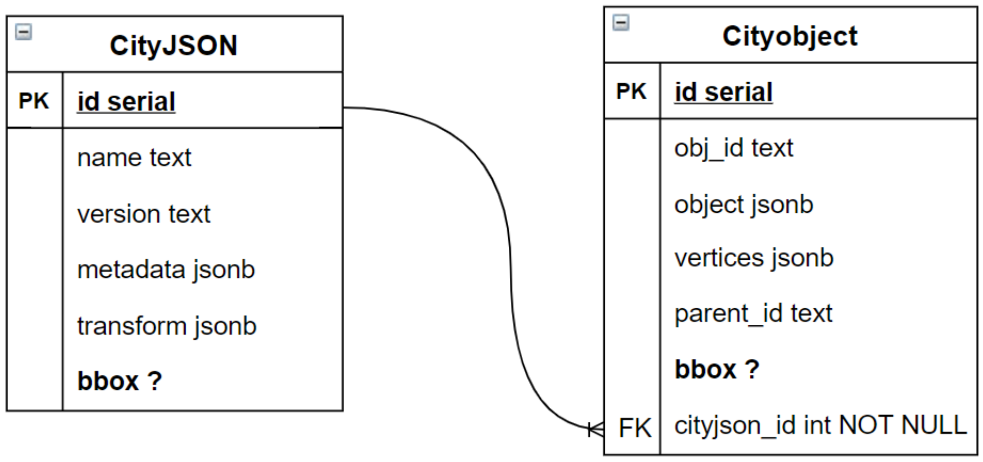


Figure 3.14: The database schema with bounding box data.

The next step is to decide how to store objects in the database. To better support the bounding box filtering, three different data types are considered:

- **geometry**: is a spatial data type provided by the extension, PostGIS. The bounding box can be represented as a WKT, which is ISO/IEC-compliant and was originally defined by the OGC [OGC, 2021]. The bounding box is a special geometry shape, for which, two corner points are sufficient for geometric representation. Instead of using `ST_GeomFromText(text WKT, integer srid)`, `ST_MakeEnvelope(float xmin, float ymin, float xmax, float ymax, integer srid=unknown)` is a better choice to simplify the expression of insertion SQL.

  To construct the SQL for bounding box filtering, the bounding box of the filter is also converted to the geometry type by `ST_MakeEnvelope` function, and the `&&` operator is used to check whether the two bounding boxes are intersected (see Listing 3.4).

```
bbox&&ST_Envelope('SRID=4326;LINESTRING(4.3743 51.8351,4.6077
    51.9671)'::geometry)
```

  Listing 3.4: Bounding box filtering with *geometry* type.

- **box**: is a built-in geometric type in PostgreSQL, which can be represented by a pair of corner points. The value of the *box* type can be specified by the syntax: `((MINX, MINY), (MAXX, MAXY))`.

  Similar to *geometry* type, `&&` operator can be applied to *box* type and the bounding box of the filter should also be converted to the *box* type (see Listing 3.5).

```
bbox&&box '((4.3743,51.8351),(4.6077,51.9671))'
```

  Listing 3.5: Bounding box filtering with *box* type.

- **array**: is a traditional data type, with which the bounding box can be represented as `[MINX, MINY, MAXX, MAXY]`.

  Different from the first two data types, `&&` operator is not applicably in the *array* data type. Instead, the intersection judgement needs to be explicitly expressed by the four values (see Listing 3.6).

```
not (bbox[1]>4.6077 or bbox[3]<4.3743 or bbox[2]>51.9671 or bbox
    [4]<51.8351)
```

  Listing 3.6: Bounding box filtering with *array* type.

An indexing mechanism can help in improving the query speed. Therefore, *GiST* indexes are applied to the bounding boxes with *geometry* and *box* types, and *B-tree* index is used in the ones with *array* type.

Another important aspect related to bounding box filtering is the CRS. The CRS can be attached to the bounding boxes with *geometry* type by assigning *SRID*, while other two data types cannot store the CRS information. Two options are available for the *box* and *array* types:

- **Same with the related CityJSON**: The bounding boxes are computed from a list of vertices, so the CRS of these bounding boxes are initially the same with the related CityJSON. In this case, the CRS is unknown to the RESTful API. Therefore, before the bounding box filtering is performed, the CRS of the CityJSON need to be queried first and then CRS conversion on the filter's bounding box may be preformed to insure that the filter and CityJSON have the same CRS.

- **WGS84**: A CRS can be pre-defined and applied to all the bounding boxes in different CityJSON datasets, for example, using WGS84. To achieve this, the bounding boxes with other CRSs need to be transformed before being inserted into the database. In this case, the RESTful API is confirmed that the CRS of the bounding boxes in the database is WGS84 and there is no need to query the CRS from the database. Still, CRS conversion on the filter's bounding box are needed when its CRS is not WGS84.

Considering that the to-be-filtered CityJSON datasets are undetermined when implementing the bounding box filtering on multiple CityJSON datasets. Therefore, the CRS information can not be retrieved before start bounding boxes on city objects. For the convenience of bounding box filtering on multiple CityJSON datasets, storing bounding boxes with WGS84 is preferable when using *box* and *array* types.

3.3.4 Attribute filtering

The semantic information at city object level is only considered in this research, which contains type and attributes properties. The type property is compulsory for each city object, thus can be stored in a static column in *Cityobject* table. However, the schema of attributes property in different city objects can largely vary. Thus, a JSONB column is chosen for flexibly storing the attributes property. In theory, the type property can also be added into the attributes property as a key/value pair, rather than storing it separately. Based on the above, there are two options for storing the type property:

- **In separate column**: The static column with *text* data type is used for storing the type property. To effectively support attribute filtering, *B-tree* index is applied.

- **In JSONB column**: The type property is stored in a JSONB column together with other attributes. The JSONB data type also supports indexing mechanism. However, if the entire JSONB column is indexed by *B-tree*, the indexing mechanism does not work with single key/value operations. Instead, *B-tree* expression index can be used to point a property, in this case, the type (see Listing 3.7).

```
CREATE INDEX btree_type ON cityobject USING BTREE (((attributes->>'
    type')));
```

Listing 3.7: Expression index on a JSONB column.

When storing type in JSONB column, the semantic information can be stored in one binary column, which provides a more compact table schema. However, PostgreSQL does not maintain statistics (such as most common values) for the column with JSONB data type, thus the query planner cannot decide which join algorithms and scan types to use for better query [Robinson, 2016]. Besides, the indexing performance on JSON column is unknown and needs to be benchmarked.

To obtain a better attribute filtering service, an overview of attribute information which contains the queryable attributes and corresponding values are sent to users through the RESTful API. To reduce the process time in the RESTful API, the information of the attributes for each CityJSON can be prepared in advance and stored in *CityJSON* table using an auxiliary column named *attribute_info* (see Figure 3.15).

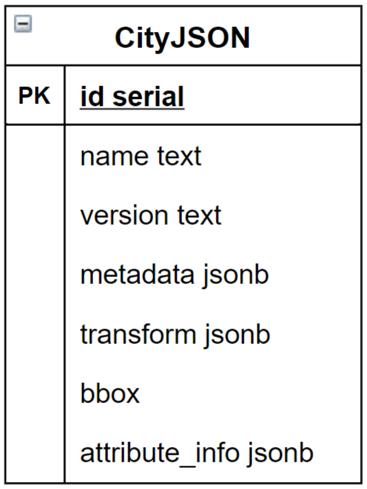


Figure 3.15: The *CityJSON* table with *attribute_info* column.

Chapter 4

Implementation

4.1 Tools and Datasets used

4.1.1 Tools

A brief overview of tools, programming languages and necessary libraries used for the proposed methodology is depicted in Figure 4.1. PostgreSQL is chosen for the database solution and manipulated by SQL. Flask is a light web framework, which can be adopted for fast prototyping a RESTful API. Jinja is a web template engine for the Flask framework, which is also a developer-friendly templating language for Python. Additionally, JavaScript is used for developing more functions (such as 3D visualization and bounding box selection) in the front end. Libraries and extension are accordingly added for the implementation.

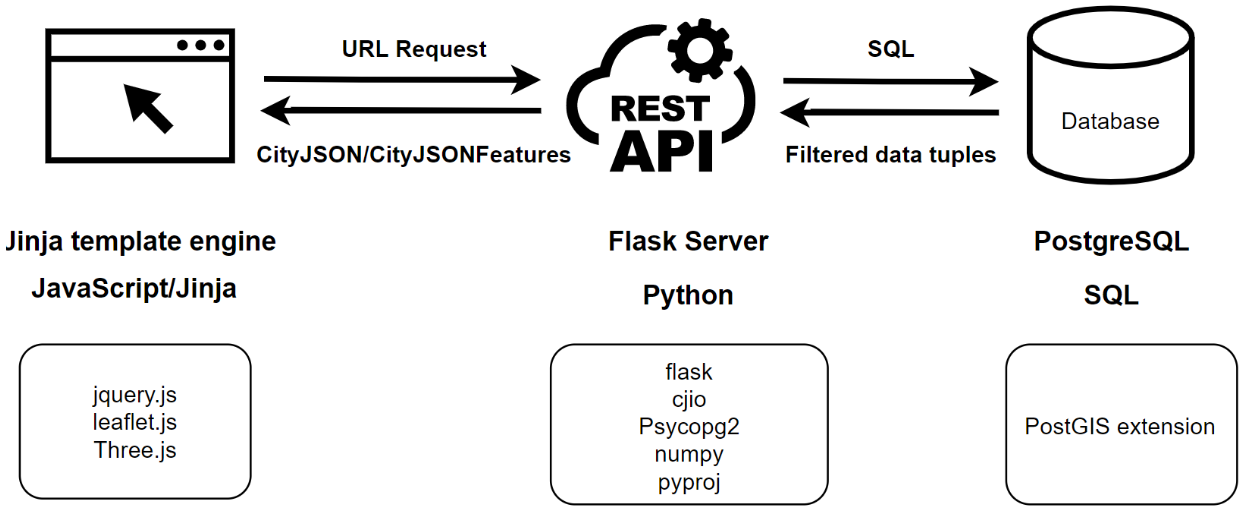


Figure 4.1: The overview of tools used.

4.1.2 Datasets

To develop and test the RESTful API, CityJSON datasets are selected from [CityJSON, 2021]. The sizes of these datasets range from 2.6 MB to 292.9 MB. To further prove the scalability of the DBMS and robustness of the RESTful API, more CityJSON datasets are collected from [PDOK, 2021], which covers the whole Netherlands and divides the CityJSON data into tiles.

4.2 Implemented prototype

The most important implementation details are described in this section which include properly storing CityJSON in database and developing a RESTful API for CityJSON datasets. The source code of the full implementation is available in GitHub.

4.2.1 Storing CityJSON in a database

Figure 4.2 shows the workflow of storing one CityJSON dataset into the database. The CityJSON file is parsed to a Python dictionary object. The properties version, metadata, tansform are stored into *CityJSON* table, and the vertices and CityObects are stored into *Cityobject* table. In addition, 2D bounding boxes and attribute information have been added to better support the RESTful API, which will be explained below.

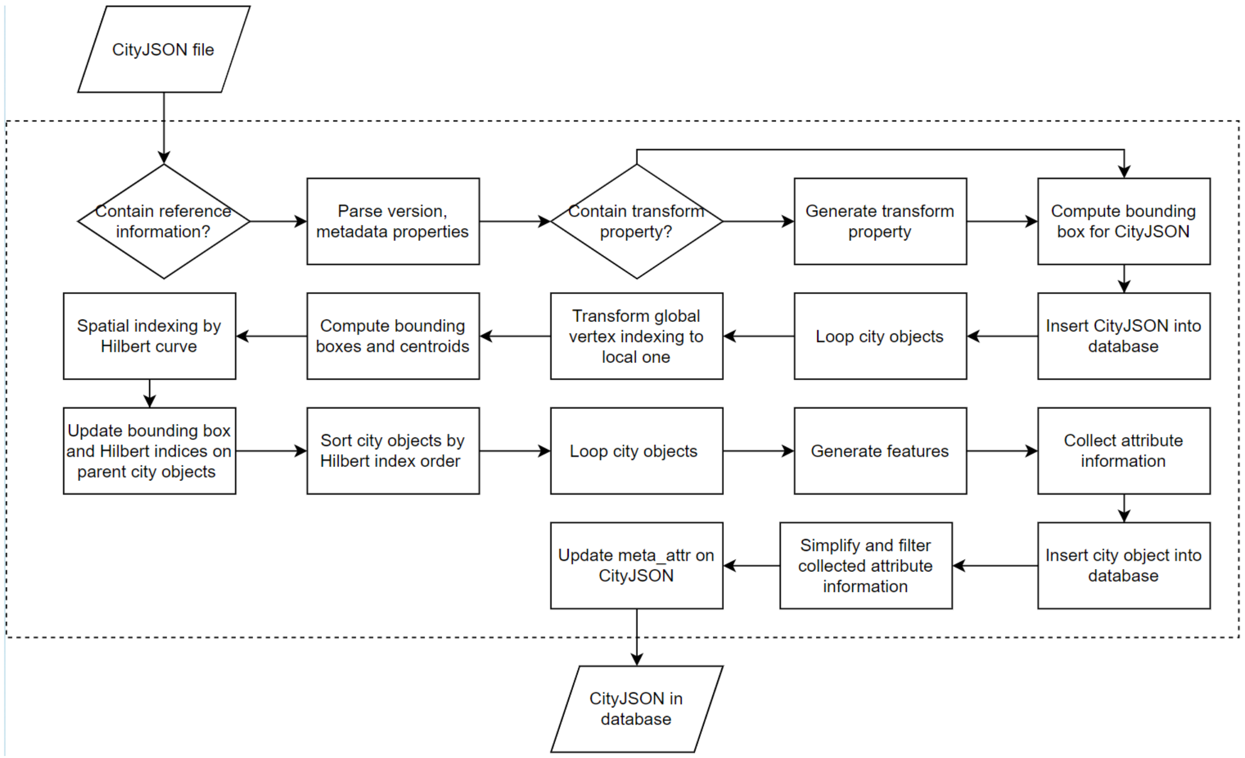

CityJSON file
Contain reference information?
Parse version, metadata properties
Contain transform property?
Generate transform property
Compute bounding box for CityJSON
Insert CityJSON into database
Loop city objects
Transform global vertex indexing to local one
Compute bounding boxes and centroids
Spatial indexing by Hilbert curve
Update bounding box and Hilbert indices on parent city objects
Sort city objects by Hilbert index order
Loop city objects
Generate features
Collect attribute information
Insert city object into database
Simplify and filter collected attribute information
Update meta_attr on CityJSON
CityJSON in database

Figure 4.2: The workflow of storing CityJSON in a database.

- **Bounding boxes**: The bounding boxes for CityJSON and city objects and computed to assist with bounding box filtering. The bounding box for CityJSON can be directly computed from the property vertices. However, there may be some unused vertices (noises) in the global vertex list, resulting in an incorrect bounding box. To solve this problem, these vertices are removed before calculating the bounding box.

  A local list of vertices is tied to each city object, which is derived from the global vertex list. Similarly, the bounding boxes for city objects can be obtained by the local vertex lists. For some city objects, whose geometry information is stored in associated city objects, the bounding boxes are computed from the child' bounding boxes.

- **Attribute information**: The attribute information for one CityJSON dataset is generated and stored in the database. The overflow of the process is given in Figure 4.3 and consists of two steps:

  - Collection of attribute information: An empty object named `attr_info` is initialised to collect attribute information. During the iteration of city objects, the property type and attributes are added into the `attr_info`. When adding new attribute data, the data will be appended to the corresponding attribute in case the attribute exists in `attr_info`. If the attribute is not a key in `attr_info`, a new key/value pair will added.
  - Filtering and simplification of attribute information: According to the rule discussed in Section 3.2.4, only attributes with numeric and string types are kept in the collected attribute information. In this respect, if the attribute is a numeric type, the minimum and maximum values will be computed to replace the collected value. For attributes with a string type, the duplicate items in the collected values will first be removed. When the number of the unique values exceed 25, the attribute will be discarded.

  In the end, the attribute information is added into the *attribute_info* column in *CityJSON* table.

4.2.2 Spatial coherence enhancement

The city objects are indexed by the Hilbert SFC and are inserted into the database by the index order to enhance the spatial coherence. The workflow of spatially indexing city objects are depicted in Figure 4.4 and contains two main steps:

- Creation of a SFC for CityJSON: A grid is generated based on the bounding box of CityJSON dataset to cover all city objects. The cell size is set to be logarithmic to the number of city objects. Then centroids of cells are indexed by Hilbert curve.

- Indexation of city objects based on the SFC: The centroids of city objects are computed from the bounding boxes. To rapidly look up the nearest cell for each city object, a KDTree

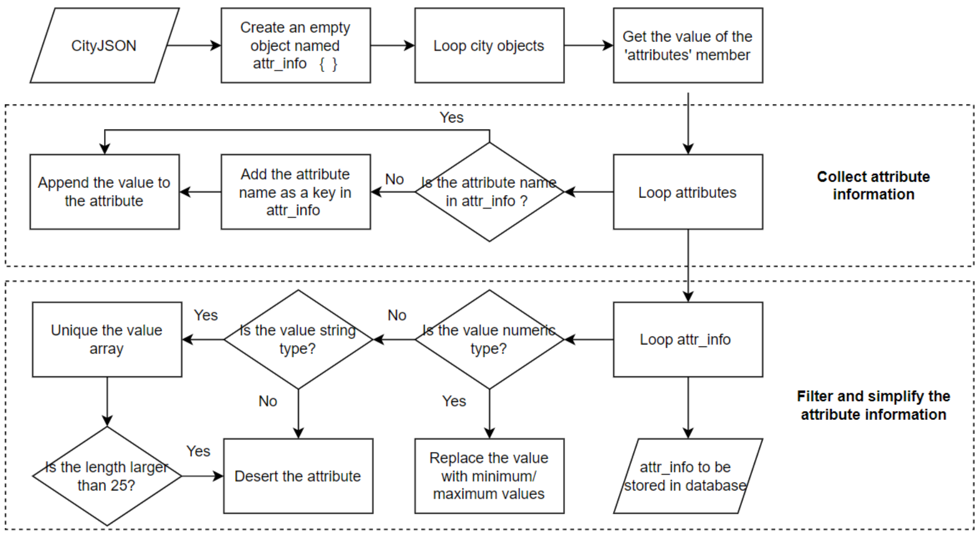

CityJSON
Create an empty object named attr_info { }
Loop city objects
Get the value of the 'attributes' member
Yes
Append the value to the attribute
Add the attribute name as a key in attr_info
No
Is the attribute name in attr_info ?
Loop attributes
Collect attribute information
Unique the value array
Yes
Is the value string type?
No
Is the value numeric type?
Loop attr_info
No
Yes
Filter and simplify the attribute information
Is the length larger than 25?
Yes
Desert the attribute
Replace the value with minimum/ maximum values
attr_info to be stored in database

Figure 4.3: The workflow of generating the attribute information for one CityJSON dataset.

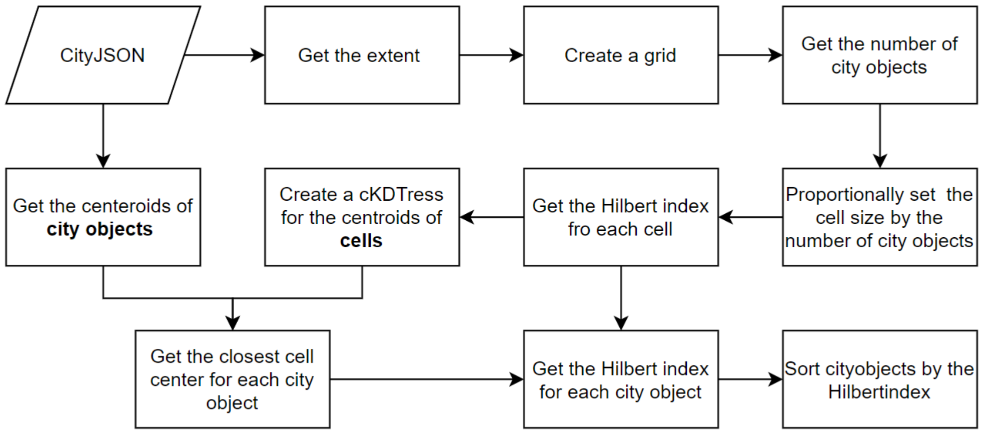

CityJSON
Get the extent
Create a grid
Get the number of city objects
Get the centeroids of city objects
Create a cKDTress for the centroids of cells
Get the Hilbert index fro each cell
Proportionally set the cell size by the number of city objects
Get the closest cell center for each city object
Get the Hilbert index for each city object
Sort cityobjects by the Hilbertindex

Figure 4.4: The workflow of indexing city objects by Hilbert space filling curve.

has been generated to index centroids of all cells. After querying the closest cell for each city object, the Hilbert index of the cell is assigned to the city object.

In the end, city objects are inserted into the database based on the Hilbert index order.

4.2.3 Access data through REST API

The overview of the implemented RESTful API is given in Figure 4.5. The resources listed in Table 3.1 have corresponding URLs with different URL parameters. The response type can be a HyperText Markup Language (HTML) web page (in default) or a JSON string that can be toggled by the parameter `f`. Next, the parameter `limit` and `offset` are for API pagination, which has been described in detail in Section 3.2.2. Filtering functions are served by appending `bbox`, `epsg` and `attrs` parameters to URLs for one or multiple CityJSON datasets.

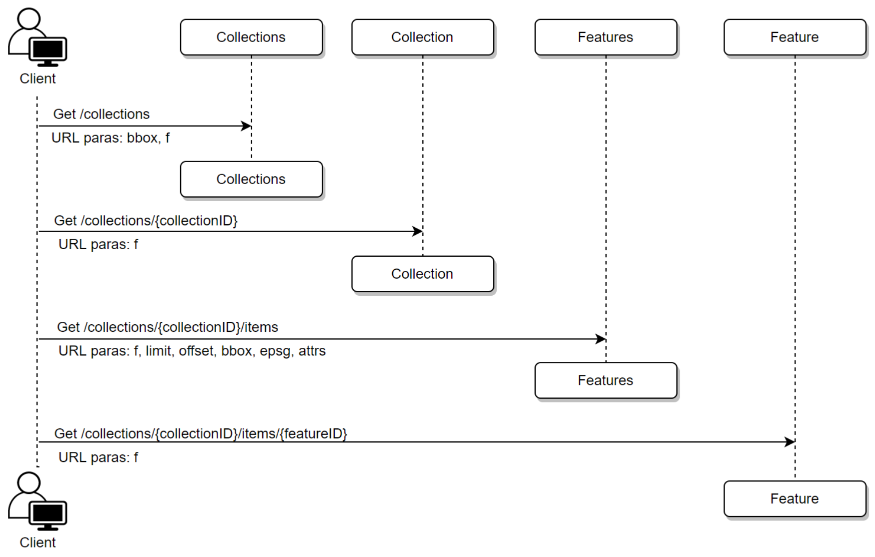

Client
Collections
Collection
Features
Feature
Get /collections
URL paras: bbox, f
Collections
Get /collections/{collectionID}
URL paras: f
Collection
Get /collections/{collectionID}/items
URL paras: f, limit, offset, bbox, epsg, attrs
Features
Get /collections/{collectionID}/items/{featureID}
URL paras: f
Client
Feature

Figure 4.5: The overview of the implemented RESTful API.

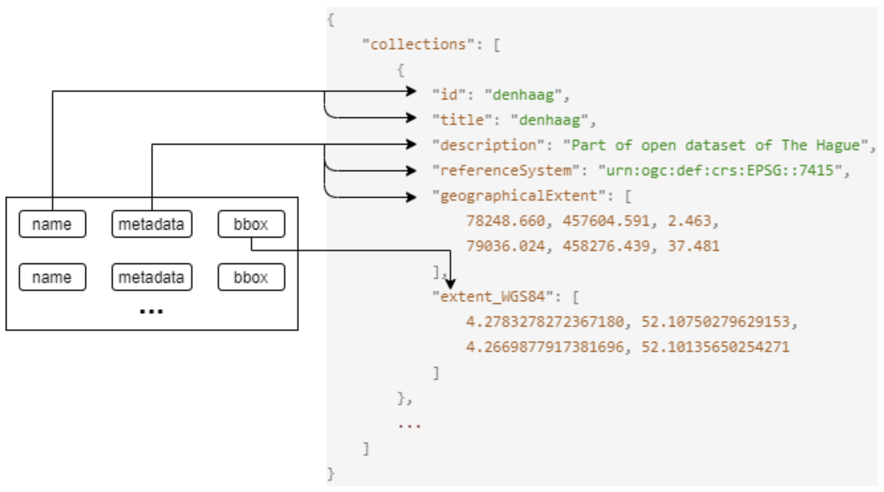
Data access with JSON response

When the parameter `f` is set to json, the response will be a JSON string. The RESTful API is developed based on the loose coupling concept, which provides great flexibility for other applications to create the customised web services. The implementation details for each resource is described as follows:

- **Collections**: The path `/collections?f=json` returns a list of CityJSON overviews that contains the identifier, title, description, reference system, 3D geographic extent with its own CRS and 2D extent in WGS 84. The *CityJSON* table is queried and each row in the results is used to construct a CityJSON overview. An example is given in Figure 4.6. The *name* from the query result is assigned to the id and title. Description, reference system, 3D geographic extent are parsed from the *metadata*. The 2D extent in WGS 84 is generated from *bbox* column.

  To implement the bounding boxes filtering on multiple CityJSON datasets, two parameters `bbox` and `epsg` are appended to the path `/collections`. The workflow of bounding box filtering on multiple CityJSON datasets is depicted in Figure 4.7. Since the auxiliary data bbox in the database is stored with CRS of WGS84, a CRS conversion is performed on the filter bounding box using Python package `pyproj` if the value of `epsg` is not 4326.

  The bounding box filtering is first carried out on *CityJSON* table. The cityjson_id, version, metadata, reference system and transform of the filtered CityJSON are collected. Note that only when the filtered CityJSON objects have the same CRS, the filtering process will continue. Then, the CRS is assigned to the metadata property of a new CityJSON object. Before constructing the transformation matrix for the new CityJSON object, the real ver-



```
{
    "collections": [
        {
            "id": "denhaag",
            "title": "denhaag",
            "description": "Part of open dataset of The Hague",
            "referenceSystem": "urn:ogc:def:crs:EPSG::7415",
            "geographicalExtent": [
                78248.660, 457604.591, 2.463,
                79036.024, 458276.439, 37.481
            ],
            "extent_WGS84": [
                4.2783278272367180, 52.10750279629153,
                4.2669877917381696, 52.10135650254271
            ]
        },
        ...
    ]
}
```

Figure 4.6: An example of constructing a CityJSON overview.

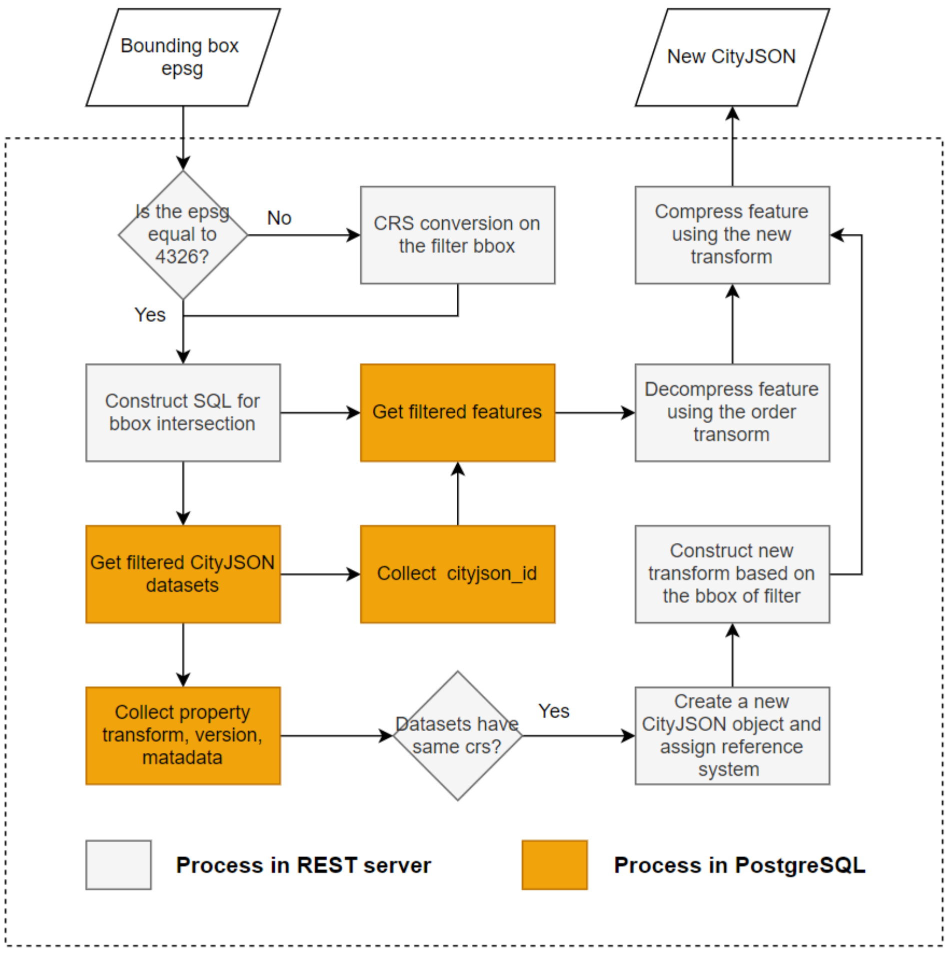

Bounding box epsg
New CityJSON
Is the epsg equal to 4326?
No
CRS conversion on the filter bbox
Compress feature using the new transform
Yes
Construct SQL for bbox intersection
Get filtered features
Decompress feature using the order transorm
Get filtered CityJSON datasets
Collect cityjson_id
Construct new transform based on the bbox of filter
Collect property transform, version, matadata
Datasets have same crs?
Yes
Create a new CityJSON object and assign reference system
Process in REST server
Process in PostgreSQL

Figure 4.7: The workflow of bounding box filtering on multiple CityJSON datasets.

tices of city objects are firstly obtained from the original transformation matrices by the following equation:

$$\mathcal{V}new_i = \mathcal{V}_i \times transform.scale_i + transform.translate_i \qquad i \in (0,1,2)\,. \tag{4.1}$$

Iterating through the list of vertices to transform each vertex can lead to a lot of overhead. Hence, it is recommended to use the vectorization ability within the Python package *NumPy* to replace explicit loops with array expressions [McKinney, 2017]. Under the same transformation rule, these vertices are transformed back to integer values and the new transformation matrix is stored in the new CityJSON object. The way of generating the new transformation matrix discussed above is based on the global list of vertices when all city objects are added into the new CityJSON object. It can be seen that this method is contradictory to the data streaming process. To better support data streaming, the new transformation matrix can be constructed based on the filter's bounding box, which has already indicated the spatial extent of the filtered results. The transform generation process for the second method consists of two steps:

- CRS conversion on filter's bounding box: The CRS of filter's bounding box can either be WGS84 or converted to WGS84 before the bounding box filtering is performed in the database. However, the spatial extent should have the same CRS with the CityJSON object. Hence, a second conversion is needed to insure that the filter's bounding box has the same CRS with the new CityJSON object.
- Construction of the properties "scale" and "translate": The scale is set to 0.001, which means that the 3 decimals will be reserved during integerization. The first two values for the "translate" property are set to the lower left corner point while the last value for Z is set to 0, since the filter's bounding box is unable to provide a vertical range.

After the new transformation matrix is prepared, the vertices for each feature is first decompressed by its previous transformation matrix and then compressed with the new transformation rule.

- **Collection**: The resource returns an overview of the CityJSON dataset identified in the path. Different from the CityJSON overview listed in the resource **Collections**, attribute information queried from the *meta_attr* column in *CityJSON* table is also added to the response.
- **Features**: Offset pagination is used to avoid long latency and internet overflow when large amounts of data are returned. Given that the response contains a complete CityJSON object instead of a list of city objects, *CityJSON* table is first queried to retrieve some necessary information in order to construct a new CityJSON object (see Listing 4.1).

```
SELECT id, version, metadata, transform
FROM cityjson
WHERE name=%s
```

Listing 4.1: Querying data in *CityJSON* table

The parameter `limit` and `offset` are parsed from the request URL and added to the SQL when querying features in a paging way (see Listing 4.2). Note that the type is limited to TOPLEVEL to avoid duplicate child city objects in the response.

```
SELECT feature
FROM cityobject
WHERE cityjson_id=%s and type In (
    TOPLEVEL)
LIMIT %s OFFSET %s
```

Listing 4.2: Querying features in a paging way

The workflow of the bounding box filtering on one specific CityJSON dataset is similar to bounding box filtering on multiple CityJSON datasets, and the main difference lies in the property transform. There is no need to construct new transform and modify the vertices when implementing bounding box filtering on one specific CityJSON dataset because all filtered city objects are under the same integerization rule.

Unlike the parameter `bbox` and `epsg`, which have static data structure, the parameter `attrs` is a JSON-based string and its content largely varies from case to case. The parameter `attrs` is first parsed and converted to a Python object using `json.loads()`, where each key/value pair represents a attribute filtering condition. Thus, a loop operation is performed on the Python object `attrs` to construct a complete SQL for attribute filtering. The filtering syntax in SQL can be different according to the attribute name, type and operation (see Figure 4.8). When the filtered data is retrieved from the database, the remaining steps are similar to the bounding box filtering.

- **Feature**: The name of the CityJSON dataset and feature id are specified in the request path. As mentioned in Section 3.3.2, features of city objects at the first and second levels are both constructed and stored in *Cityobject* table. Therefore, there is no additional operation in SQL to query the associated city objects (see Listing 4.3).

```
SELECT feature
FROM cityobject
WHERE cityjson_id=%s AND obj_id=%s
```

Listing 4.3: Querying one feature

Data access with HTML response

An interactive working demo is developed to show the applicability of the RESTful API. Data transmitted from the RESTful API is filled onto the web page by *Jinja* template Engine. *Spectre.css*

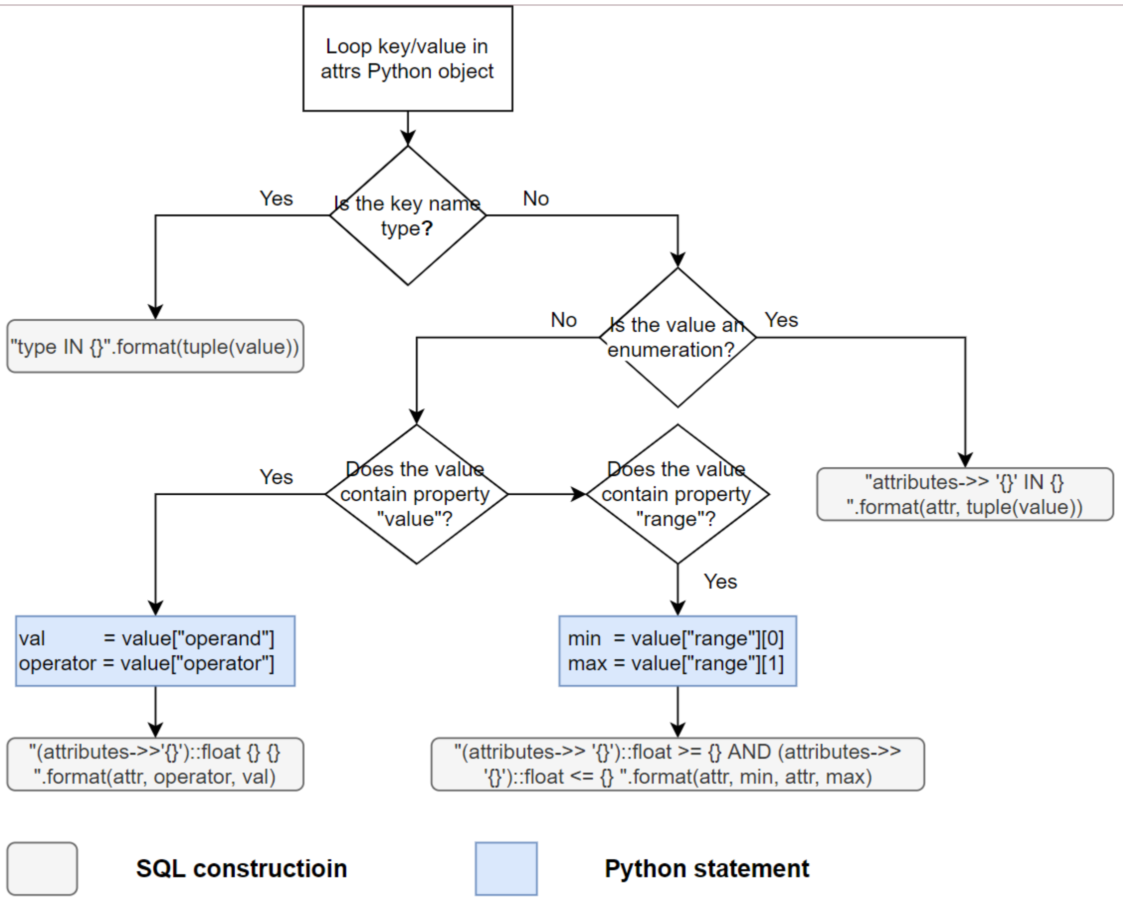

Loop key/value in attrs Python object
Is the key name type?
Yes
No
"type IN {}".format(tuple(value))
Is the value an enumeration?
No
Yes
"attributes->> '{}' IN {} ".format(attr, tuple(value))
Does the value contain property "value"?
Yes
Does the value contain property "range"?
Yes
val = value["operand"]
operator = value["operator"]
min = value["range"][0]
max = value["range"][1]
"(attributes->>'{}')::float {} {} ".format(attr, operator, val)
"(attributes->> '{}')::float >= {} AND (attributes->> '{}')::float <= {} ".format(attr, min, attr, max)
SQL constructioin
Python statement

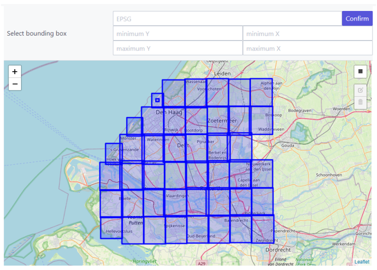
Figure 4.8: An example of constructing a CityJSON overview.

CSS Framework is employed to provide better web page rendering effect. Additionally, *leaflet.js* and *Three.js* are used to visualize 2D and 3D objects, respectively. Three additional functions have been added into the demo to improve the user experience:

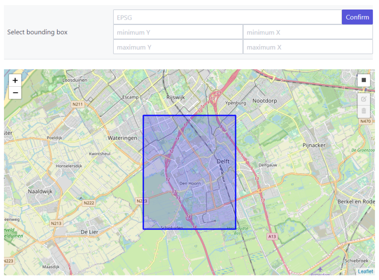
- **2D bounding box**: When accessing the resource *Collections* and *Collection*, 2D bounding boxes are visualized under the CRS of WGS84. To enhance the spatial perception, OpenStreetMap tiles are added into the map as base layer (see Figure 4.9).

  Users can request bounding box filtering on one or multiple CityJSON datasets by typing coordinates of two corner points of the bounding box, and epsg value into the table. Another option is to select a ROI by drawing a rectangle over the data layer (see Figure 4.10). Once the ROI is selected, the coordinates of two corner points will be automatically filled into the table and the epsg value is set to 4326, which offers great convenience for users to request bounding box filtering.

- **3D city model**: The browser-based visualization of 3D city models can provide a clear and direct data overview to users and does not require users to download specific software or plug-ins. *ThreeJS* is an easy-to-use 3D library, which can be used to visualize 3D city

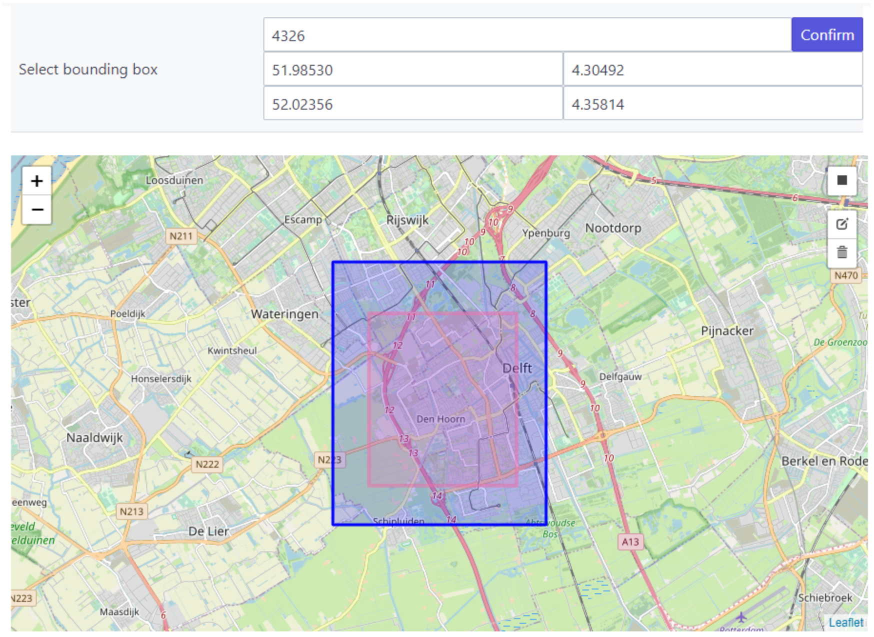


(a) 2D bounding boxes in the resource *Collections*



(b) 2D bounding box in the resource *Collection*

Figure 4.9: Visualization of 2D bounding boxes.



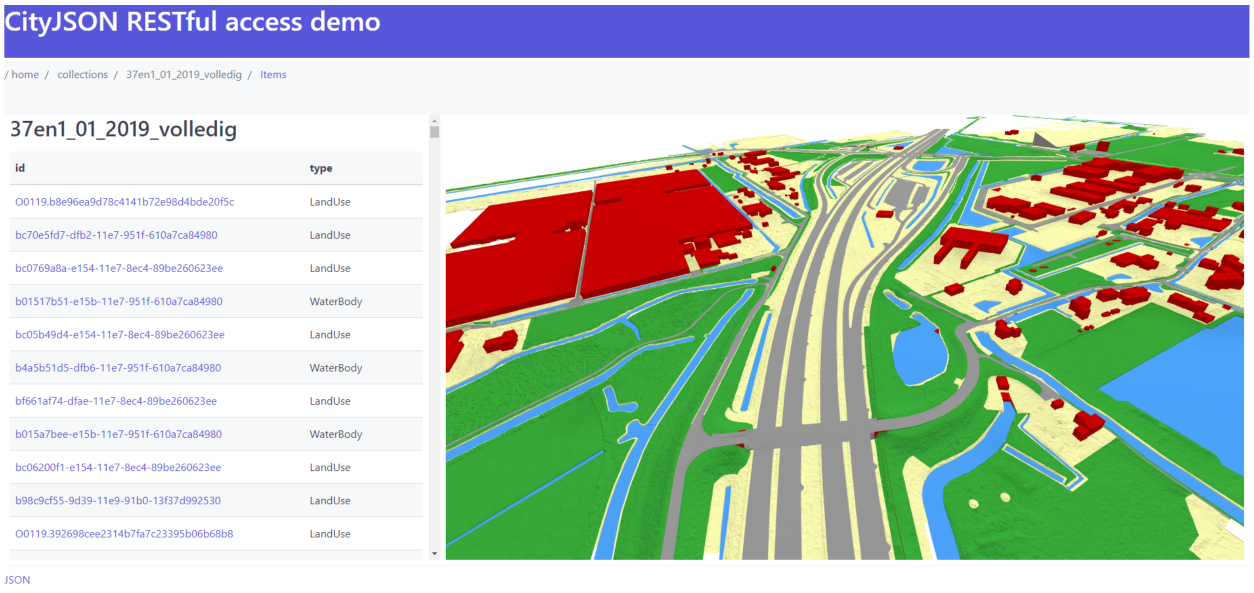
Select bounding box

4326	Confirm
51.98530	4.30492
52.02356	4.35814

Figure 4.10: The drawing tool for selecting a ROI.

models on a web page. Each city object is added to the viewer as a triangulated *Mesh*, where the type of city object is indicated by the material color (Figure 4.11).

- **Attribute**: To provide a clear overview of attributes contained in one CityJSON dataset, components in *Spectre* CSS Framework are used to display the attribute information on the web page. In addition, graphical attribute filtering operations are also offered in the web page to relieve the effort for the user to construct a JSON string with attribute filtering conditions. An example is given in Figure 4.12. When the user presses the confirmation button of attribute filtering, the JSON string is constructed using *jQuery* and sent to the RESTful API.



CityJSON RESTful access demo

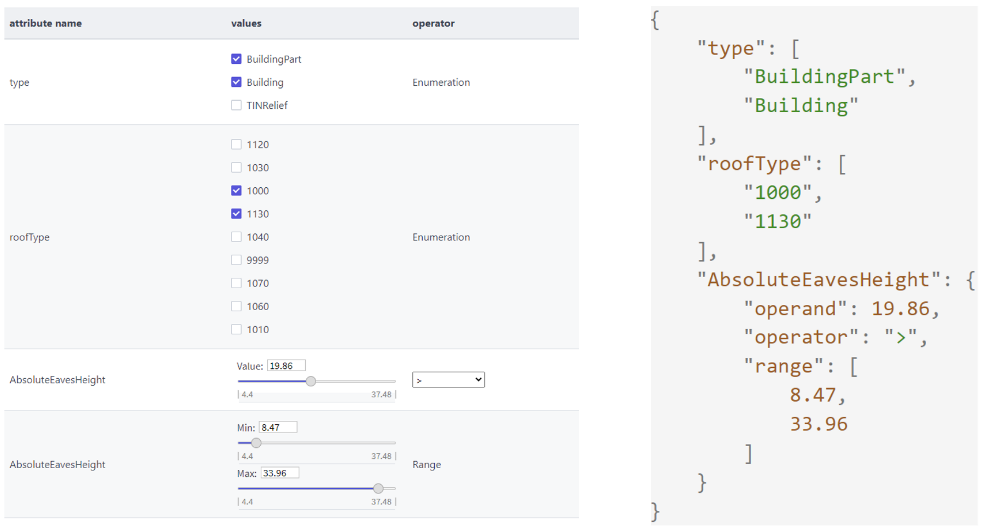
/ home / collections / 37en1_01_2019_volledig / Items

37en1_01_2019_volledig

id	type
O0119.b8e96ea9d78c4141b72e98d4bde20f5c	LandUse
bc70e5fd7-dfb2-11e7-951f-610a7ca84980	LandUse
bc0769a8a-e154-11e7-8ec4-89be260623ee	LandUse
b01517b51-e15b-11e7-951f-610a7ca84980	WaterBody
bc05b49d4-e154-11e7-8ec4-89be260623ee	LandUse
b4a5b51d5-dfb6-11e7-951f-610a7ca84980	WaterBody
bf661af74-dfae-11e7-8ec4-89be260623ee	LandUse
b015a7bee-e15b-11e7-951f-610a7ca84980	WaterBody
bc06200f1-e154-11e7-8ec4-89be260623ee	LandUse
b98c9cf55-9d39-11e9-91b0-13f37d992530	LandUse
O0119.392698cee2314b7fa7c23395b06b68b8	LandUse

JSON

Figure 4.11: The 3D viewer for 3D city models in the RESTful API.



attribute name	values	operator
type	☑ BuildingPart ☑ Building ☐ TINRelief	Enumeration
roofType	☐ 1120 ☐ 1030 ☑ 1000 ☑ 1130 ☐ 1040 ☐ 9999 ☐ 1070 ☐ 1060 ☐ 1010	Enumeration
AbsoluteEavesHeight	Value: 19.86 4.4 37.48	>
AbsoluteEavesHeight	Min: 8.47 4.4 37.48 Max: 33.96 4.4 37.48	Range

```
{
    "type": [
        "BuildingPart",
        "Building"
    ],
    "roofType": [
        "1000",
        "1130"
    ],
    "AbsoluteEavesHeight": {
        "operand": 19.86,
        "operator": ">",
        "range": [
            8.47,
            33.96
        ]
    }
}
```

Figure 4.12: An example of constructing attribute filtering conditions.

4.2.4 Data streaming

The large amounts of filtered data are chunked into small ones and sent to the user in a streamed way. Data transmission experiences two phases: from the DBMS to the RESTful API and from the RESTful API to the web browser. Both phases need extra process to implement the data streaming.

DBMS solution

Two methods of chunking data in the filtered tuples are proposed in Chapter 3 and the implementation details are discussed below:

- **Paginating the query**: The two parameters `limit` and `offset` are used for paginating the query. The `limit` is equal to the chunk size and `offset` is set to 0 for the first query. After the RESTful API constructs the data based on the first query result, the value of `offset` is added by the chunk size and then the RESTful API queries the database again. The whole process is repeated until there is no data queried from the database.

- **Streaming the query results**: Different from the first method, all the filtered tuples are generated on the database side by one query. The next step is to fetch a small part of data in succession until all filtered tuples are transmitted to the RESTful API. To implement this, the *cursor* size is set to the chunk size. Then a *while* loop is used to continuously fetch a limited amount of data from the database (see Listing 4.4).

```
cur.arraysize = CHUNK_SIZE
while True:
    rows = cur.fetchmany()
    if not rows:
        break
    ...
```

Listing 4.4: Streaming the query results

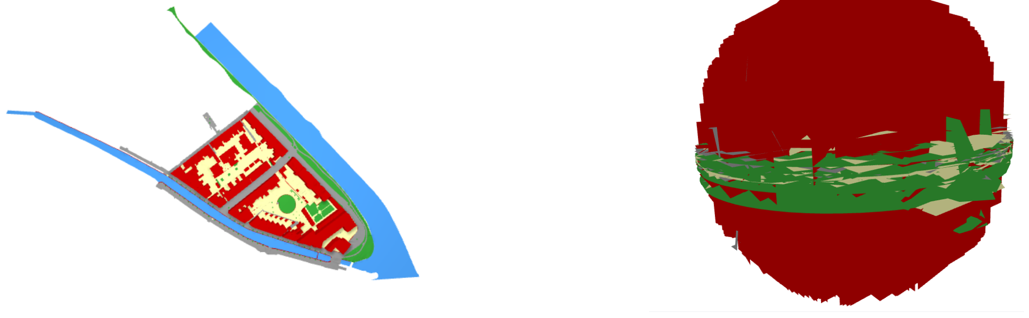
REST API solution

When a small portion of filtered tuples arrives at the RESTful API, the API immediately constructs a set of CityJSONFeatures. To provide a faster first response, once a CityJSONFeature is generated, it will be asynchronously sent to the user through the *Generator* function in Python. Note that the basic information including reference system and transformation matrix are sent to the user before CityJSONFeatures.

4.2.5 Visualization with streaming

When the features are streamed to the browser, they are added to the 3D Web viewer separately. However, individual visualization of each feature will result in incorrect visualization (see Figure 4.13). This is because the coordinates of the newly added object are first normalized to fit the position of the camera. In this case, each feature is normalized based on its own shape, which causes the mixed topological relation between features.

To solve the above problem, a unified transformation matrix for normalization should be constructed before adding feature into the scene. The transformation matrix can be calculated



(a) Adding features as a whole (CityJSON) (b) Adding features individually

Figure 4.13: The incorrect visualization when adding features individually

from the spatial extent. Let max and min are the two corner points of the spatial extent. Then the transformation matrix can be obtained by the following equation:

$$\begin{aligned} center_i &= (max_i - min_i)/2 \quad i \in (0,1,2), \\ scale &= 1/Max(center_x, center_y, center_z). \end{aligned} \tag{4.2}$$

The spatial extent can be accurately calculated from the global vertex list. However, the global vertex list cannot be obtained during data streaming. To overcome this, the spatial extent is obtained directly from the "geographicalExtent" property in "metadata", which is sent along with other basic information before streaming features. The "geographicalExtent" property can be inferred from the filter's bounding box when filtering multiple CityJSON datasets, which is the similar process of constructing "transform" property when filtering on multiple CityJSON datasets.

When the browser receives the "MetadataCityJSONFeature" object, the transformation matrix for normalization is computed before visualization. Note that the filter's bounding box does not provide the vertical extent. Thus, the vertical value in the "geographicalExtent" property is set artificially and not correct. Therefore, when determining the scale value, the Z-value of the center is ignored to avoid the inaccurate normalization.

From Figure 4.14 we can see that features are visualized correctly during steaming. In addition, storing city objects by the order of SFC enhances the spacial coherence and makes the visualization better, especially for visualization with data streaming.

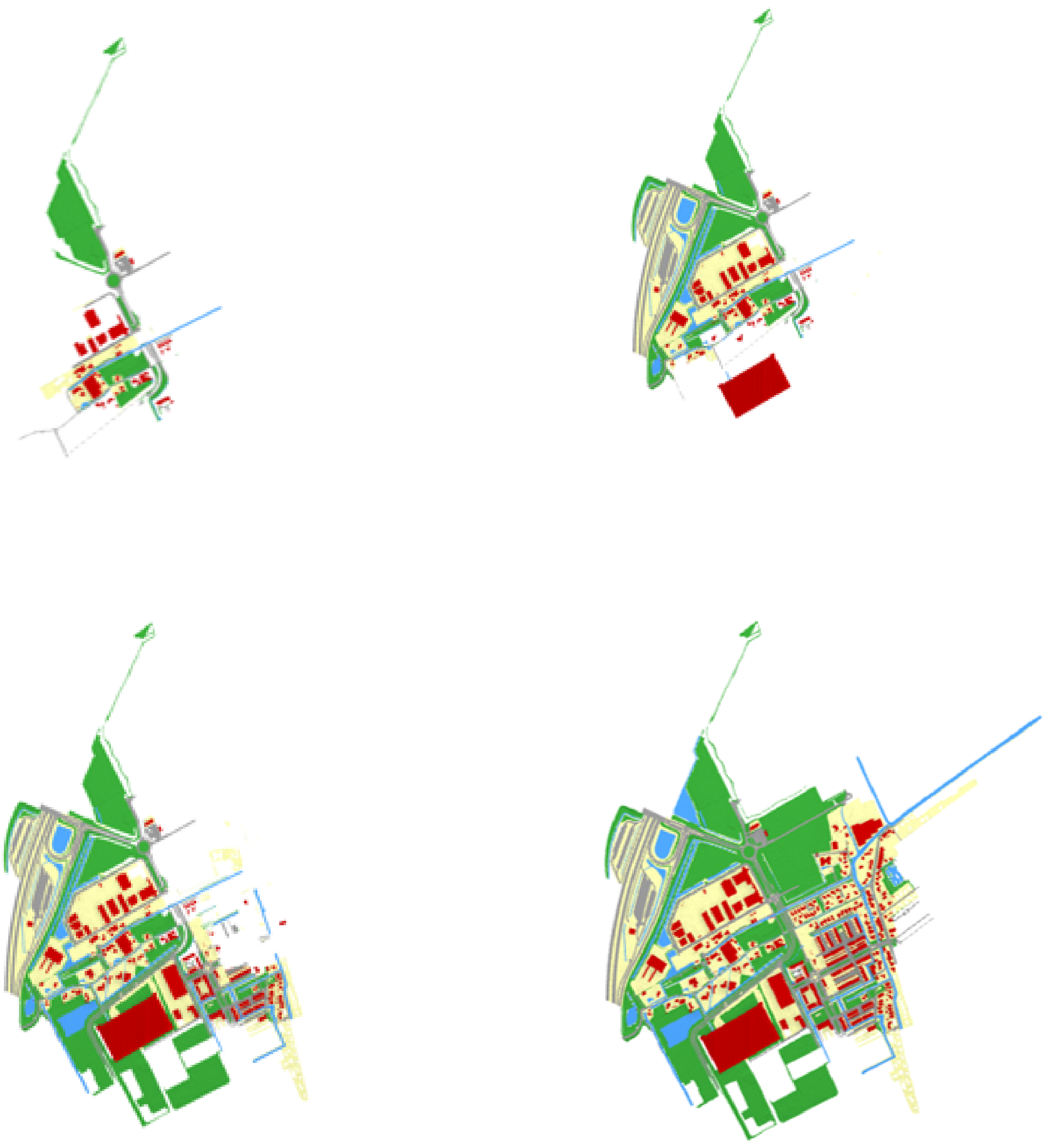


Figure 4.14: The visualization with streaming

Chapter 5

Benchmarking and analysis

This chapter evaluates the design of the database schema, the performance of the streaming methods and the difference between DBMS and a file system regarding the efficient data access through the RESTful API. The benchmarking concerning the topics above have been performed and the results are discussed in this chapter.

5.1 Tools and Datasets

5.1.1 Software

JMeter is a load testing tool for measuring service performance and is especially useful for web applications [Apache JMeter™, 2021]. It can be used to assess the performance of database schema by JDBC database connections and to evaluate the performance of the RESTful API by HTTP Requests. In addition, JMeter supports clearing cache after each HTTP request and throttling outgoing bandwidth to simulate the other network speed, thereby making the benchmark results more reliable.

The throttling is implemented by modifying the *.properties* file of JMeter (see Listing 5.1) and the *cps* (bytes/s) is set to 6912000 (equal to 54 Mbit/s), which is the same bandwidth with WIFI 802.11a/g.

```
httpclient.socket.http.cps=6912000
httpclient.socket.https.cps=6912000
```

Listing 5.1: Simulating the WIFI network speed in JMeter

5.1.2 Hardware

Table 5.1 gives the specifications of the used server for the benchmarking. The specification will also influence the web performance, especially the CPU and memory when processing large amounts of data.

Processor	Intel® Core™ i5-8400 CPU
Memory	8 GB
Storage	500 GB SSD
OS	Windows 10 Home 64-bit

Table 5.1: Specification of the computer used for benchmarking.

5.1.3 Datasets used

The benchmarks on very small datasets (less than 20 MB) are trivial and are susceptible to unexpected factors. Thus, only CityJSON dataset of *Zurich_Building_LoD2_V10* has been selected from [CityJSON, 2021]. Additionally, the developed RESTful API should be able to serve as many CityJSON datasets as possible. Therefore, another 62 CityJSON datasets are collected from [PDOK, 2021], in which 5 are containing various city objects, and the others only contain buildings (see Figure 5.1) . The overview of the chosen CityJSON datasets is summarized in Table 5.2. All the chosen CityJSON datasets are inserted into database for benchmarking, which contains 2106663 city objects in total.

(a) CityJSON containing only buildings

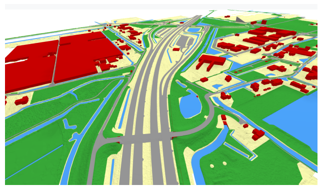

(b) CityJSON containing various city objects

Figure 5.1: The CityJSON datasets downloaded from [PDOK, 2021].

File name	File size (MB)	City objects	Characteristics	Source
Zurich_Building_LoD2_V10	286.1	198699	Multi-part buildings	CityJSON [2021]
37en1 ... (57)	20.2 - 133.8	10585 - 63386	Single part buildings	PDOK [2021]
37en1_volledig ... (5)	438.4.5 - 2218.9	19239 - 106417	Various city objects	PDOK [2021]

Table 5.2: The overview of CityJSON datasets used in the benchmarking.

5.2 Overview of benchmarking methodology

A brief overview of benchmarking methodology is depicted in Figure 5.2 and consists of three main aspects:

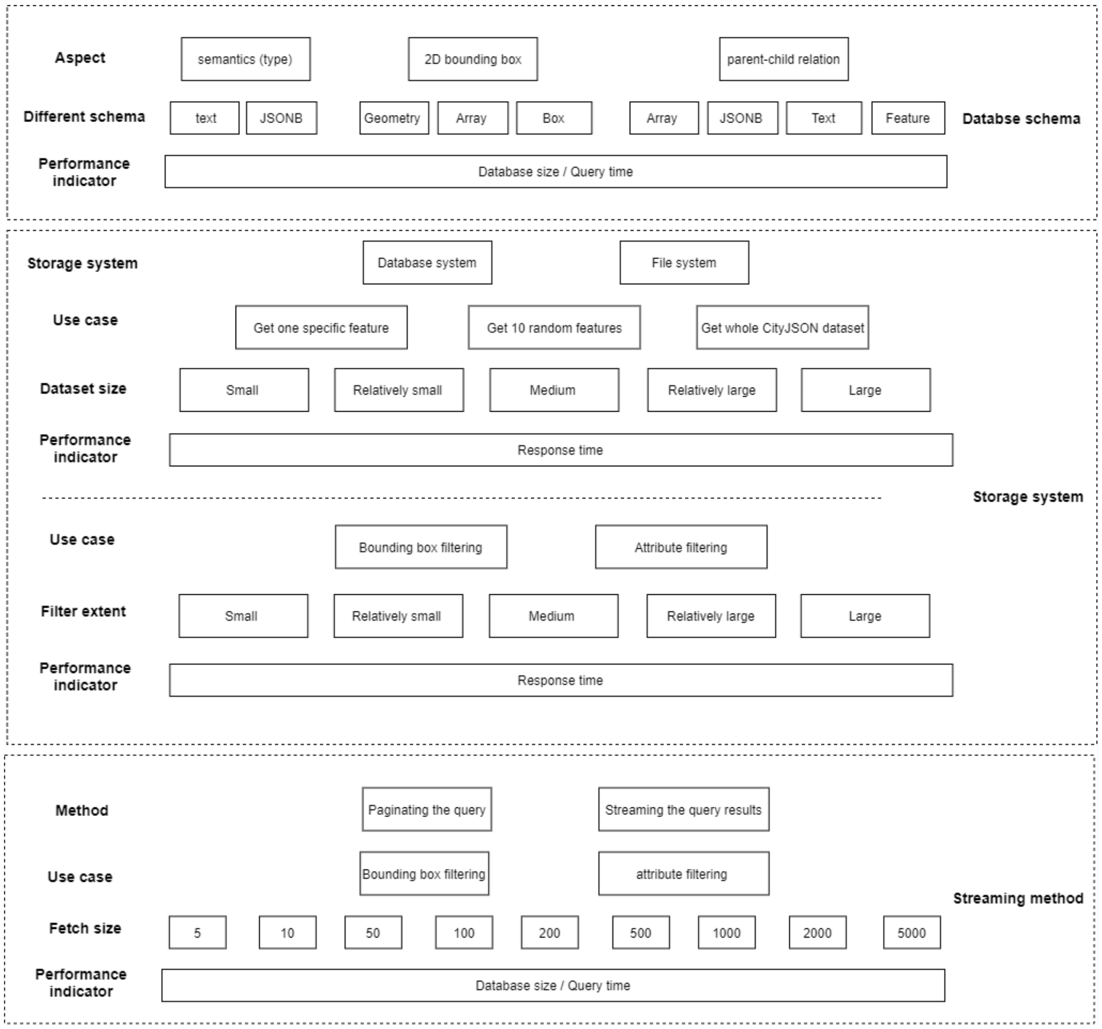


Figure 5.2: An overview of benchmarking methodology.

- **Database schema**: Different ways of storing the type and 2D bounding boxes of city objects, and the hierarchical relation are benchmarked to determine the trade-off between storage size and query speed.

- **Storage system**: Database solution can provide better scalability, but it takes extra time for the RESTful API to assemble city objects retrieved from the database the to a new CityJSON object. Four use cases are designed to comprehensively test which storage solution is suitable for which use case.

- **Streaming method**: Compared to streaming the query results, paginating the query is expected to serve a quicker first response but require a longer time to load all the data. The performance of two data streaming methods is measured with different fetch sizes from the database.

5.3 Database schema

Database schema plays an important role in data access efficiency. To better support the data filtering and streaming process, the type value, 2D bounding box and parent_child relation are derived from the CityJSON dataset and explicitly added into database schema. Different ways of storage for the three parts are proposed in Chapter 3 and the performance is benchmarked and discussed, respectively.

5.3.1 Metrics

An important step in benchmarking is to define the quantitative metrics. Different storage schema will lead to different data storage sizes and query performance. Regarding the storage size, it also consists of three aspects:

- Schema size: This refers to the sum of all table sizes. Hence, the ways of storing CityJSON in the database directly influence the schema size.
- Database size before extra indexes: The database size includes information other than the pure data in tables, including all associated indexes, TOAST space, free space map, and visibility map [PostgreSQL, 2021c].
- Database size after extra indexes: The different index choices depend on the column types (e.g. array or JSONB), which have an indirect impact on database size and query performance.

In addition to the data types, certain constraints in the database schema design also plays an important role in the storage size and query performance. For example, normalization constrain is used to reduce redundancy and improve consistency. However, this may increase the number of tables and reduce the performance of data queries.

Hence, it can be seen that there are usually trade-offs between storage size and query time. Therefore, these two factors are chosen to indicate the quality of the database schema.

5.3.2 Semantics

The property "attribute" and "type" store the semantic information at the city object level, which will be use during attribute filtering process. The property "attribute" is stored in a JSONB column, which is explained in Section 2.4.1. The type value can either be stored in the JSONB column as an additional key/value pair or it can be separately stored in a *text* column. Two database schema are created and different indexing types are used to improve the efficiency of data queries (see Listing 5.2). Note that only the type member in the JSONB column is indexed.

```
-- index type in JSONB column
CREATE INDEX ON schema1.cityobject USING BTREE (((attributes->>'type')));
-- index type in a separate column
```

```
CREATE INDEX ON schema2.cityobject USING BTREE (type);
```

Listing 5.2: Indexing type values in PostgreSQL.

Storage size

From Figure 5.3 we can see that the schema size is smaller (6.8%) when storing property "type" in a JSONB column than in a separate *text* column. The results prove the compression ability of JSONB data type. Before extra indexes are added to improve the *type* query, the database size is slightly larger when using JSONB, which indicates that additional mechanisms and information are stored in PostgreSQL to support the JSONB data type. By computing the difference between the schema size and the database size before indexing, using JSONB needs 10% more extra space than using *text*. After indexing the *type*, the database size with a separate *text* column exceeds the one with JSONB column. This can be caused by a more complicated indexing mechanism used in the *text* type.

To conclude, the difference in storage size between the two schema is trivial and does not reveal that one is much superior to the other one.

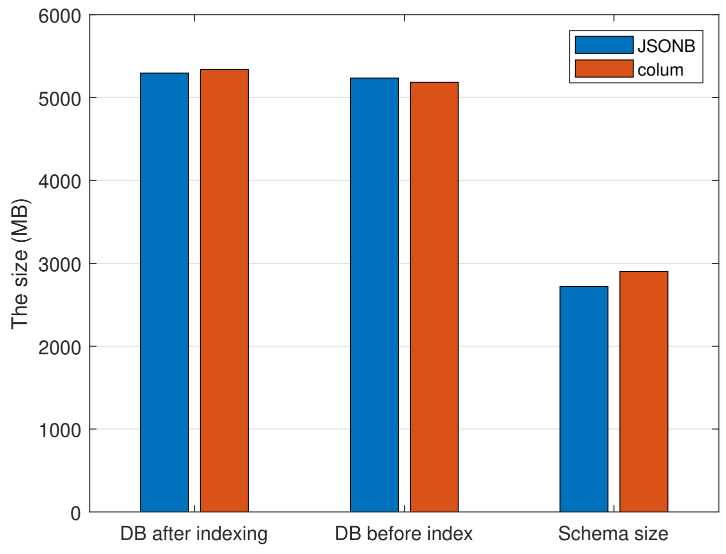


Figure 5.3: The different storage sizes of storing property "type" in a JSONB column or in a separate *text* column.

Query performance

Two use cases are designed to test the query performance: retrieving all the city objects with "BuildingPart" type and retrieving all the city objects at the first level. Additionally, the `EXPLAIN PLAN` SQL syntax is used to check the detailed information of the execution plan.

Figure 5.4 illustrates the difference of query time for finding all the city objects with "BuildingPart" type. Although a separate column offers quicker query time, the difference is insignificant and both query performance is acceptable (less than 3%).

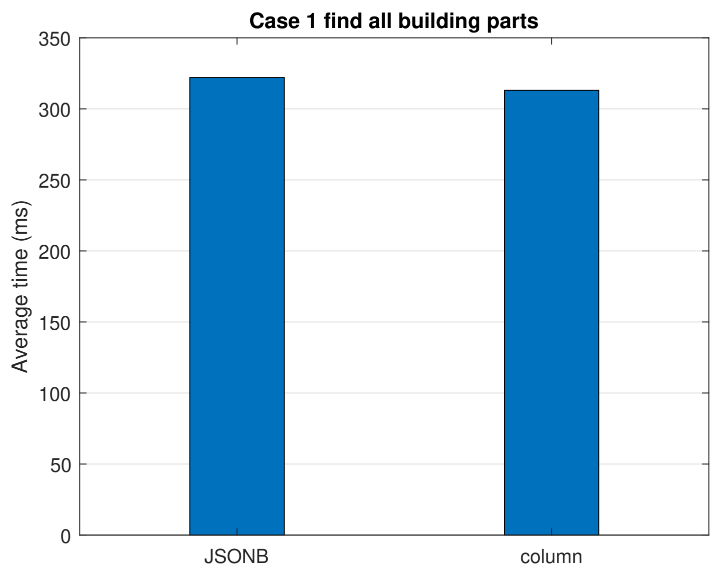


Figure 5.4: Use case 1: Retrieving all the city objects with "BuildingPart" type.

As discussed in Chapter 4, to avoid duplicate sub-city objects, the first-level city objects will be retrieved first. Hence, the second use case is created to test whether the database schema can better support the RESTful API when a small part of city objects are randomly selected or bounding box filtering is performed.

The results of query time are given in Figure 5.5. We can see that the database schema with a separate column performs much better than the database schema with a JSONB column (shrinking 40.7% of the time). To find the reason, the detailed execution plans are inspected, which show that the indexing mechanism was not executed on the database with a separate *text* column. Instead, each row is sequentially scanned until the end row when the SQL syntax `LIMIT` is not used. This is because the indexing mechanism requires an extra time of seeking and most city objects are at the first level. As a result, the query planner for the database with a separate *text* column abandon the indexing mechanism. However, the query planner for the database with a JSONB column still applies the Bitmap heap scan for a large subset of the city objects, resulting in poor query performance.

The reason why the query planner can make a better choice with traditional data types such as the text is that PostgreSQL stores statistics about the distribution of values with traditional data types in each column, such as the most common values [PostgreSQL, 2021a]. However, JSONB type lacks statistical information, as a consequence, the query planner cannot decide which join algorithms, join orders and scan types to use for better query [Robinson, 2016].

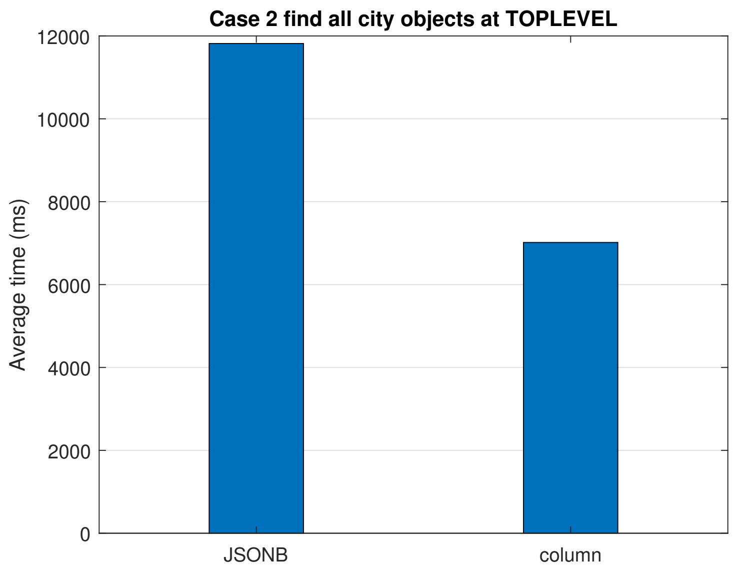


Figure 5.5: Use case 2: Retrieving all the city objects at the first level.

All in all, the traditional data types offer better query performance and do not obviously increase the database size. Therefore, storing the "type" value in a separate *text* column is a better choice. Additionally, the results from the first case present the good performance of JSONB data type when a small subset of city objects are queried. This indicate that storing the other semantic information in a JSONB column can support the attribute filtering function in the RESTful API.

5.3.3 Bounding box

An extra column is added into the database schema for storing the 2D bounding boxes. Three data types are proposed and the corresponding storage occupancy is theoretically analyzed as follows:

- **geometry**: The function `ST_MakeEnvelope()` returns a geometry represented by a WKT with double-precision (see Listing 5.3). Therefore, one bounding box requires 10 x 8 80 bytes.

```
POLYGON((84840.6220026873 437417.294006594,
        84840.6220026873 443799.077006594,
        90208.2200026873 443799.077006594,
        90208.2200026873 437417.294006594,
        84840.6220026873 437417.294006594))
```

Listing 5.3: A 2D bounding box represented by a WKT with double-precision

- **box**: The box type store 2 sets of coordinate data with double-precision. Hence, it also requires 4 X 8 32 bytes to store a 2D bounding box.

- **array**: An numeric array with four values is created and the precision is user-specified. To maintain the same precision with the first two types, double-precision is chosen. The expected storage occupancy is 4 X 8 32 bytes.

In conclusion, the schema size with the geometry type is expected to be much larger than with the other two types.

Storage size

Figure 5.6 presents the different storage sizes of storing 2D bounding boxes in a column with geometry or array or box types. When storing 2D bounding boxes with geometry data type, the schema size is noticeably smaller than with the other two types, which is contradictory to the prediction. This is because the storage is optimised by the in-built compression mechanism in PostgreSQL with large geometries [Baston, 2018]. Instead of storing a WKT in the schema, a Well-known binary (WKB) is used to compress the schema size.

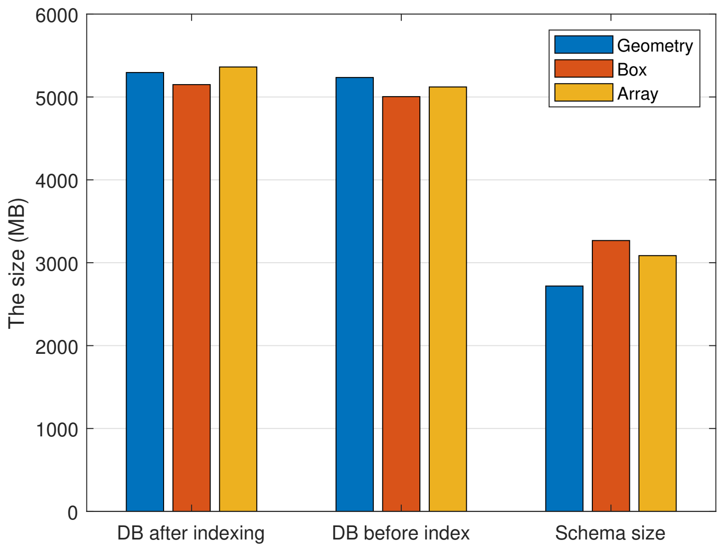


Figure 5.6: The different storage sizes of storing 2D bounding boxes in a column with geometry or array or box types.

After indexing 2D bounding boxes, the database size with 2D bounding boxes stored in an array column exceeds the other two. The possible reason is that the spatial indexing is performed on the 2D bounding boxes with geometry and box type. However, the intersection judgement for 2D bounding box with the array type needs to be explicitly expressed by the four values. Therefore, four separate indexes are created, which increase the storage overhead.

Query performance

The use case in this query performance measurement is to filter the city objects outside of the area shown in Figure 5.7. Figure 5.8 represents the results of query performance. Compared with the other two types, the query speed of the box type is almost twice as fast. The array and geometry types serve the similar query performance. The difference between the database size and schema size with geometry type is more obvious than with the other two, which indicates that performing operations on data with geometry type is complex, and more additional mechanisms are used to support the query. The box can be regarded as a special and simple geometric type, so the operations on the box data can be simplified.

To conclude, although storing 2D bounding boxes in a column with box type has the largest schema size, the total database size is the lowest. More essentially, the query performance is far better than the other two types. Hence, box type is chosen to store 2D bounding boxes.

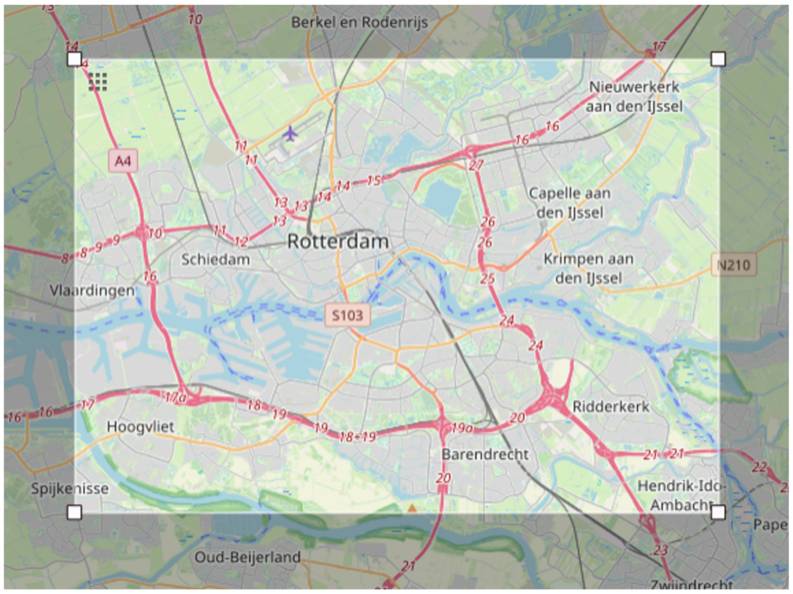


Figure 5.7: The selected area of bounding box filtering.

5.3.4 Parent-child relation

Four different methods of storing parent-child relation are proposed in Section 3.3.2. Three methods explicitly store the hierarchical relation by a parent_id column (foreign key) or an array column or a JSONB column. The forth method is to construct and store features in advance. Then the parent-child relation is indirectly reserved in those features.

One possible issue related to the fourth method is that the city objects at second level are redundantly stored in the database, which may result in a larger storage size. On the other hand, there is no additional operation to query the associated city objects in DBMS and to construct

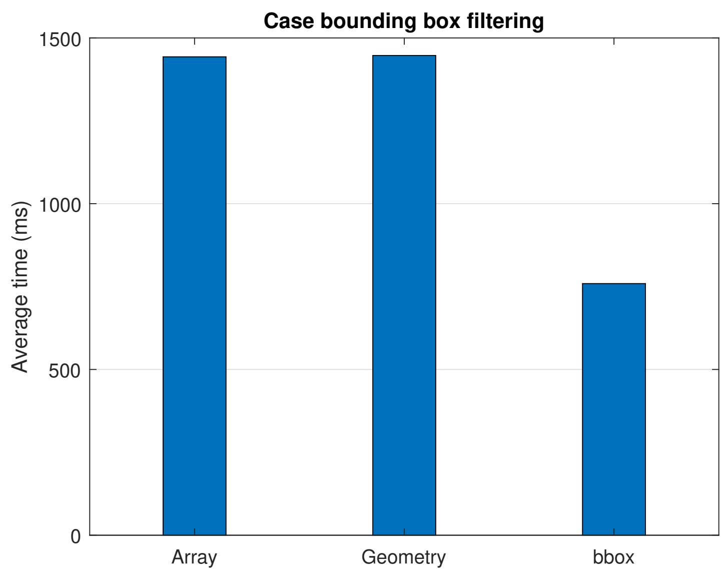


Figure 5.8: The results of query performance related to 2D bounding boxes.

the features in the RESTful API. Hence, the fourth method can theoretically serve better query and web performance.

Storage size

From Figure 5.9 we can see that the fourth method (feature) has unexpectedly lowest schema size while other three methods have similar schema sizes. Note that the JSONB solution simply stores the array as a JSON using `JSON.dumps()` during data insertion process and there is almost no size difference between the array solution and JSONB solution. This indicates that the JSONB does not apply compression on every kind of data.

Indexes are according added to improve the efficiency of querying associated city objects in the three explicit solutions and the database sizes slightly increase by the indexes. It is noticeable that the feature solution has the largest database size even without any extra indexes.

However, the results are contradictory to the previous prediction about the fourth method. One possible reason of the insignificant redundancy influence on storage size is that only one (out of 62) CityJSON dataset has sub-city objects. To better inspect the redundancy influence, only *Zurich* dataset is inserted into the database. The result is given in Figure 5.10. When using the fourth method, the database size increases by 37% compared to the foreign column method. It shows that the fourth method will lead to great storage burden when most city objects have associated city objects.

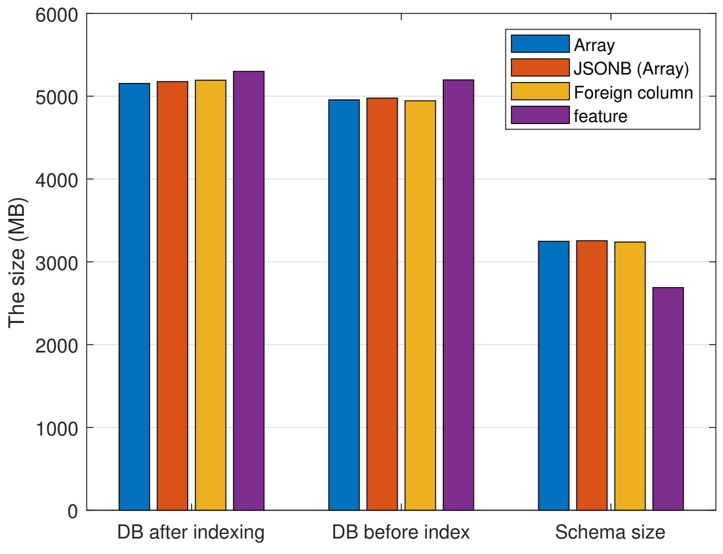


Figure 5.9: The different storage sizes of storing parent-child relation in several ways.

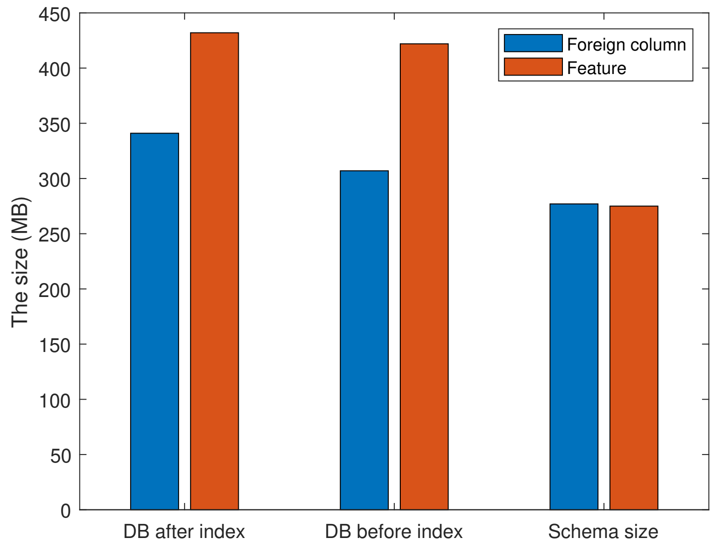


Figure 5.10: The different storage sizes of storing parent-child using *Zurich* dataset.

Query performance

Theoretically speaking, it is very easy to find the sub-city object ids when the parent-child relation is stored in the database using the three explicit methods, while the feature method struggles to accomplish this task. However, finding sub-city objects are not the ultimate goal for the RESTful API. Instead, the RESTful API should be able to return the complete sub-city objects in an efficient way. Additionally, during the data streaming process, the sub-city objects should be clustered with their parents. Thus, a sorting operation by the parent_id is necessary to perform the clustering task.

Based on the above discussion, the use case used in this performance measurement is to retrieve all city objects in the *Zurich* dataset since it is the only dataset that contains multi-part buildings. In addition, an extra column indicating the main id needs to be appended in the query results for the RESTful API to differentiate the features. Moreover, the associated city objects should be assembled together using a sort operation on the main id to support data streaming process.

The results of query performance measurement is given in Figure 5.11. The feature solution provides the best query performance while the JSONB solution has the poorest performance. The significant difference between the feature solution and the other three solutions is mainly caused by the sort operation. Sorting the query results is a resource-intensive and time-consuming process [Winand, 2021], because the complete query results must be read before a sort operation starts. Thus, the sorting can not be implemented in a pipelined way, which is contradictory to data streaming and problematic when handling a large amount of data. The array and JSONB solutions perform worse than the parent_is solution. This is because that `unnest ()` and `jsonb_array_elements_text()` functions are respectively used to flatten multiple arrays or JSONB to a set of values in the array and JSONB column. This extra process leads to a performance penalty.

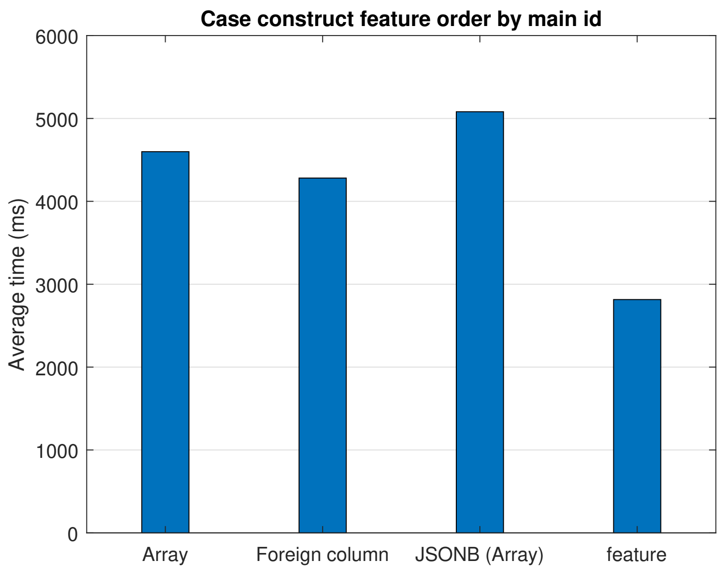


Figure 5.11: The results of query performance related to parent-child relation.

All in all, pre-constructing and storing features has the best query performance. However, the redundancy issue will become obvious when handling city objects that have associated city objects at the second level. In addition, some future work may still require the explicit parent-child relation, for example, returning parent city objects according to the city objects at the second level. Hence, adding a parent_id column with *text* type can be the most conservative and reliable way.

5.4 Storage system

The benchmarking performed in this section aims to prove the better scalability, efficiency and flexibility of data access in the DBMS over the file-based system. The designed use cases can be grouped into two categories: data access without filtering and data access with filtering. 7 (out of 63) datasets with different sizes are used in the first group, and *30dz2* and *Zurich* datasets are used in the filtering use cases based on the bounding box and attribute, respectively. To comprehensively measure the filtering performance, several HTTP requests with various filtering ranges are designed for the data access with filtering.

5.4.1 Metrics

The response time is the performance indicator for checking which storage solution can better support the use cases in the RESTful API. The response includes several phrases, which depends on the storage system type:

- DBMS: After the RESTful API receives the HTTP request, SQLs are constructed to retrieve data from the database. Then the query results on the database side are sent to the RESTful API. After new CityJSON object is generated, it is sent to the user.

- File system: After the RESTful API receives the HTTP request, the whole CityJSON file in the file system is read into the RESTful API. After data processing, new CityJSON object is generated and sent to the user.

5.4.2 Use case: access to one specific city object

The CityJSON ID and feature ID are identified in the request URL to access a specific *Feature*. The benchmarking results are illustrated in Figure 5.12. As the dataset size increases, the response time of accessing data from the file system greatly increases. Unlike DBMS, data irrelevant to the final result is also read to REST API from the file system, resulting in a waste of memory in API. With a larger dataset, memory waste becomes more serious. Moreover, due to the lack of an indexing mechanism, a loop operation is used in the API to find the specific city object. This lays a great burden to the CPU and takes a longer time to complete this task.

All in all, with the help of the built-in query and index mechanism, the database system is better than the file system in querying specific city objects.

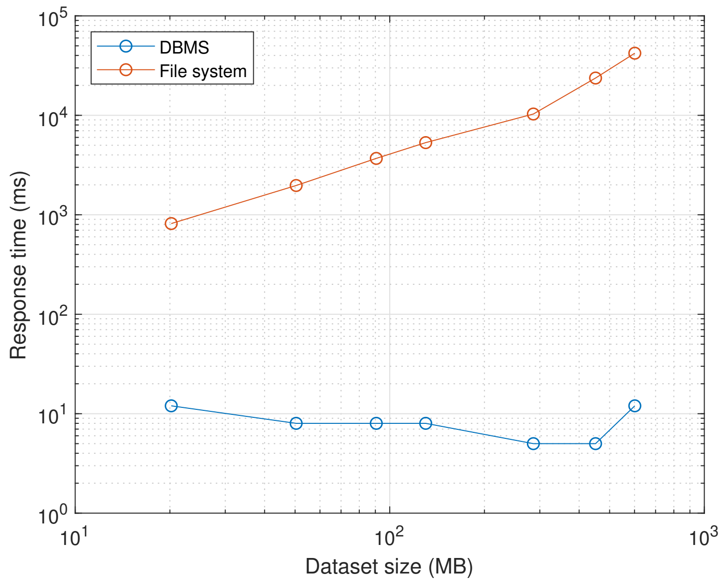


Figure 5.12: The benchmarks of access to one specific city object

5.4.3 Use case: access to 10 random city objects

The use case is to assess the performance of API pagination mechanism when using different storage systems. The parameter `limit` is set to 10, and `offset` is 0 by default. From Figure 5.13 we can see that the database system provides a superior and stable performance, while the performance of the file system is poor when dealing with large CityJSON datasets. When the dataset size ranges from 20 to 600 MB, the response time slightly fluctuates between 10 to 120 milliseconds with a database system. However, the response time with a file system increases from 600 milliseconds to 1 minute. This will become unacceptable from a user perspective, when the dataset size further increases.

Briefly to conclude, due to the lack of scalability, the file system cannot well support the API paging mechanism during data access. Thus, a database system is preferred to handle this use case.

5.4.4 Use case: access to a whole CityJSON dataset

In this use case, there is no extra data process for the file system. The CityJSON dataset is read by the RESTful API, and then directly sent to the user. However, for DBMS, city objects from the original CityJSON file need to be retrieved and then assembled to a CityJSON object in the RESTful API.

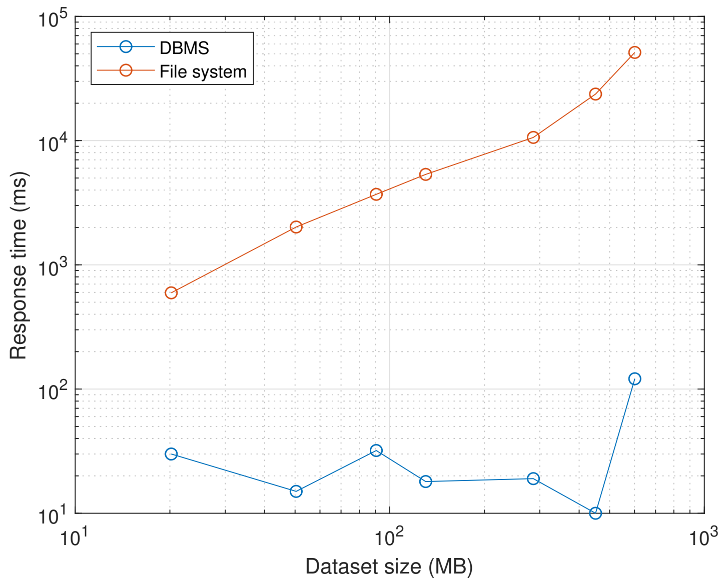


Figure 5.13: The benchmarks for access to 10 random city objects

The benchmarking results are given in Figure 5.14. When the dataset size increases, the response time takes longer due to the limited network bandwidth. It can be seen that accessing a whole CityJSON dataset from the DBMS has a higher growth rate of response time than the file system. This is because the CityJSON object is split into individual city objects with the local vertex lists. Thus, it needs extra time for the RESTful API to combine those city objects into a CityJSON dataset.

As the dataset size reaches 100 MB, the difference start enlarging. For example, when the dataset size is 286 MB, the response time (130 seconds) with DBMS is 1.38 times that (94 seconds) of the file system and when the dataset size is 600 MB, it becomes 1.70 times (404 and 238 seconds, respectively). This is because of the CPU and memory limitation. The database connection and query is also a resource intensive process, which requires high CPU and memory usage. In addition, the redundant vertices are stored in the memory before the redundant removal process. The redundant vertices occupy additional memory space and the removal process also requires CPU consumption.

5.4.5 Use case: bounding box filtering

The *30dz2* dataset of size 130 MB is used in this use case. 7 HTTP requests with different filter bounding boxes are designed to evaluate the filtering performance of the DBMS and file system.

Figure 5.15 represents the benchmarking results. As the filtering range increases, a larger dataset will be returned to the user. Therefore, the response time increases accordingly. It is

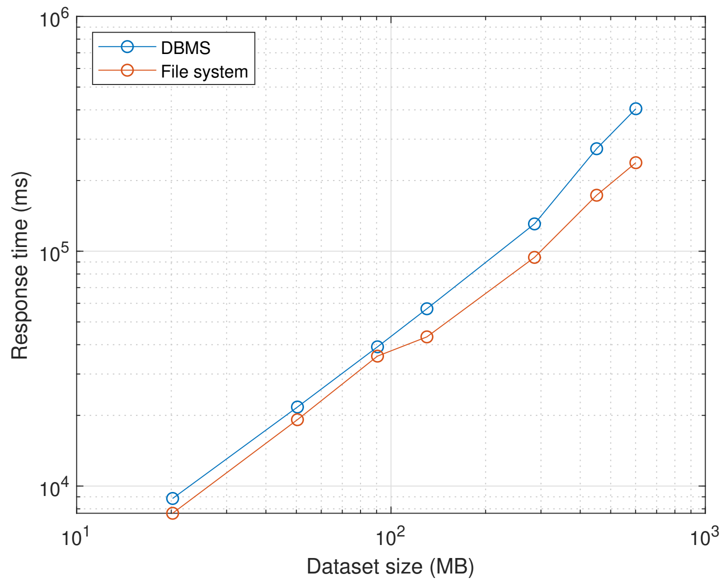


Figure 5.14: The benchmarks of access to a whole CityJSON dataset

noticeable that the database solution provides better performance. The main reason for this is that the spatial index mechanism is applied to support fast data query in DBMS. On the contrary, each city object is scanned when using the file system, which will cause performance degradation. However, as the filter range increases, the difference shrinks because the performance of indexing mechanism becomes trivial when a large portion of city objects are selected from one CityJSON dataset.

It can be concluded that accessing a whole CityJSON dataset from the file system has better performance that from the DBMS.

5.4.6 Use case: attribute filtering

The attribute filtering is based on the attribute "class" in the *Zurich* dataset. To utilize the indexing mechanism in the DBMS, the attribute "class" is indexes as follows:

```
CREATE INDEX ON api.cityobject USING BTREE (((attributes->>'class')));
```

The benchmarking results shown in Figure 5.16 are quite similar to the use case of bounding box filtering. With the help of the index, the DBMS is able to quickly find the expected city objects. Around 80% city objects (about 230 MB) in *Zurich* dataset is retrieved in the final HTTP request. Thus, we can expect that when more city objects are selected, the file system will have a better performance over the DBMS according to the trend.

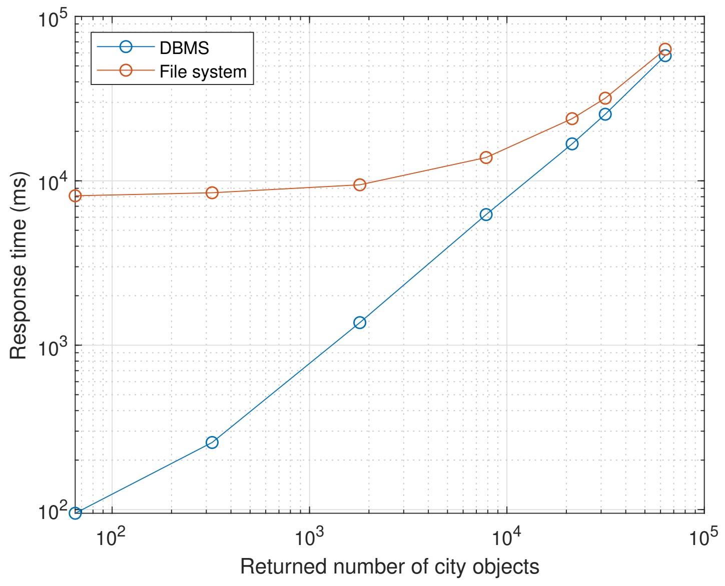


Figure 5.15: The benchmarks of bounding box filtering using two storage systems.

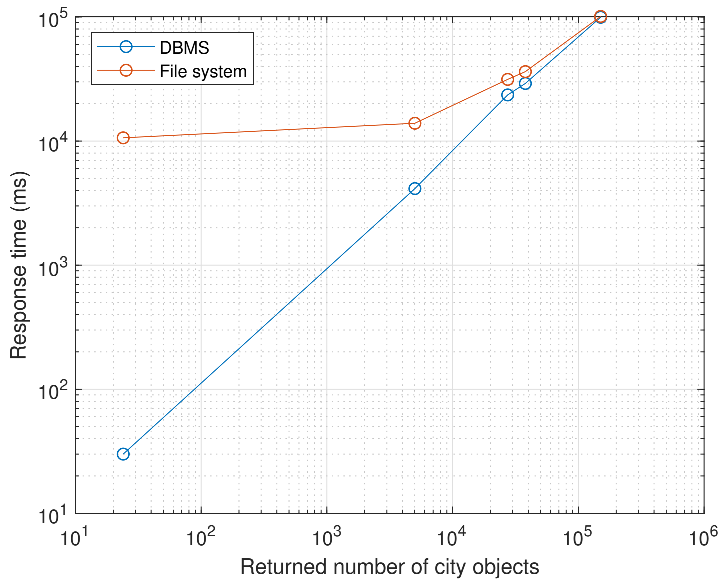


Figure 5.16: The benchmarks of attribute filtering using two storage systems.

5.5 Streaming method

Paginating the query and chunking the query results are the two methods proposed in Section 3.2.5 to implement data streaming from the DBMS to the RESTful API. Bounding box filtering on *Zurich* dataset and attribute filtering on *37en2_volledig* dataset are the two use cases designed for benchmarking and comparing the streaming performance of the two methods. In addition, the performance of not streaming data from the DBMS to the RESTful API is also benchmarked as the baseline.

5.5.1 Metrics

The metrics used for measuring streaming performance is different from the database schema, and consists of three aspects:

- The first response time: This refers to the time when the user firstly receives a CityJSON-Feature. A well-known statistic shows that 53% of web users will abandon a web page if it takes more than 3 seconds to start loading [Kirkpatrick, 2016]. Hence, the first response time is chosen as an important performance indicator.
- The total response time: This refers to the time when the user receives all the requested data. The total response time includes the data retrieval time in the DBMS, the process time in the RESTful API and data transmission time.
- The correctness rate of the returned data: The correctness rate refers to the rate between the number of non-divided CityJSONFeatures and the ideal number of CityJSONFeatures. One CityJSONFeature may be divided into two when the associated city objects are not fetched from the database at the same time (see Figure 5.17). The splitting issue is avoided in the pagination method because every paginated query collects the complete associated city objects. However, in the second method, all the city objects are selected on the database side, then chunked into small groups with a fixed number.

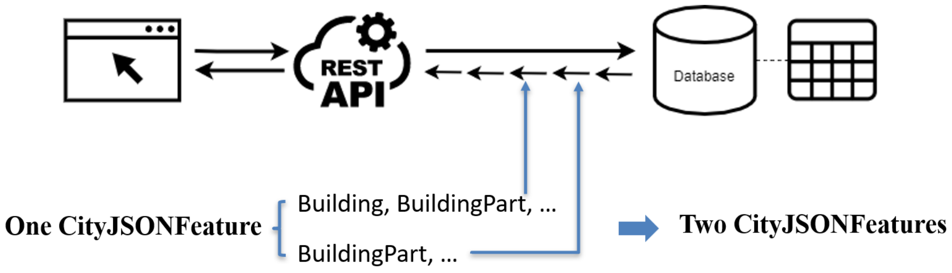


Figure 5.17: The phenomenon of divided CityJSONFeature .

5.5.2 Use case : bounding box filtering on *Zurich* dataset

The use case is to filter the city objects outside of the region shown in Figure 5.18. The expected results contain 34267 CityJSONFeatures and 136418 city objects in total. The HTTP request sent by JMeter is http://127.0.0.1:5000/collections/Zurich_Building_LoD2_V10/items/?bbox=8.47424,47.36766,8.60916,47.41647&epsg=4326.

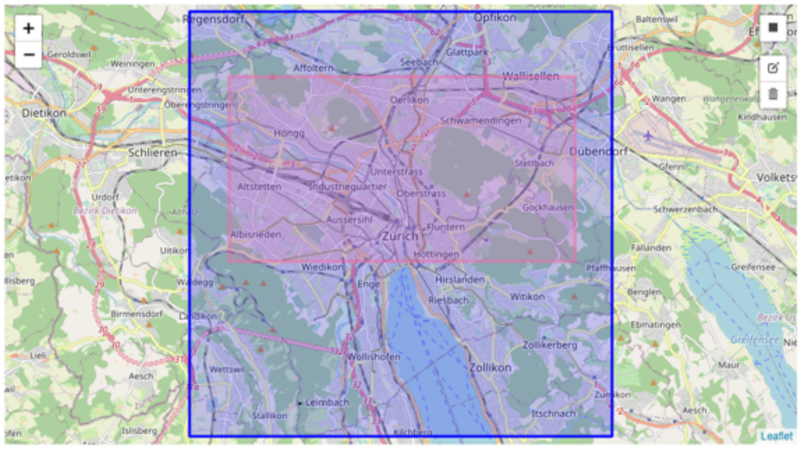


Figure 5.18: The selected area (in pink) of bounding box filtering for measuring data streaming performance.

Baseline results

Figure 5.19 shows the baseline results for this use case. When the data is not streamed from the DBMS, the RESTful API needs to wait until all the filtered tuples are transmitted, and then start constructing CityJSONFeature. Hence, it takes very long time (around 16 seconds) for the clients to receive the first CityJSONFeature. The client will receive all the CityJSONFeature with a size of 199.4 MB after about 1 minute (the total response time).

Method 1: Paginating the query

Figure 5.20 represents the data streaming performance during bounding box filtering with the method of paginating the query. The first response time stays below 400 ms when the fetch size is smaller than 1000 and starts dramatically increasing when the fetch size is larger than 1000. However, the total response time shows the opposite pattern. When the fetch size is set to a small value, the total response time is extremely long. This is because the database is connected and queried too many times, which is a time-consuming and resource-intensive process. When the fetch size is larger than 200, the total response time becomes acceptable. Note that the fetch size here refers to the number of city objects at the first level, and all the associated city objects at the second level will be queried and sent along to the RESTful API. Hence, the correctness rate is always 100%.

Method 2: Streaming the query results

The streaming performance is illustrated in Figure 5.21. Even when the fetch size is very small, the first response time is around 4.5 seconds. This is due to that fact that the database is connected and queried only once, and all the filtered results are retrieved on the database side. Therefore, if the SQL is complex or the filtered results contain large amounts of data, it will take

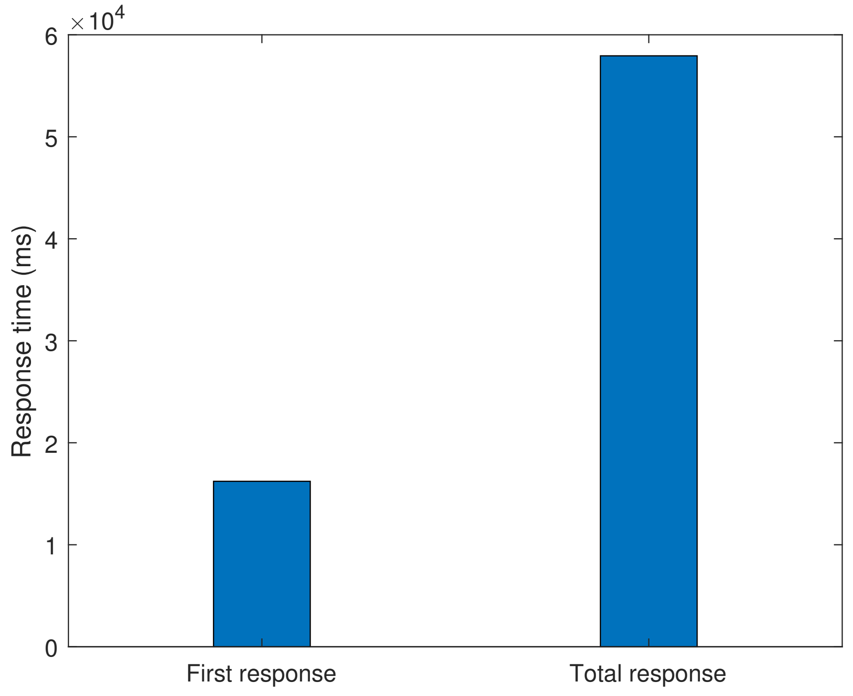


Figure 5.19: The performance during bounding box filtering without streaming from the DBMS to the RESTful API.

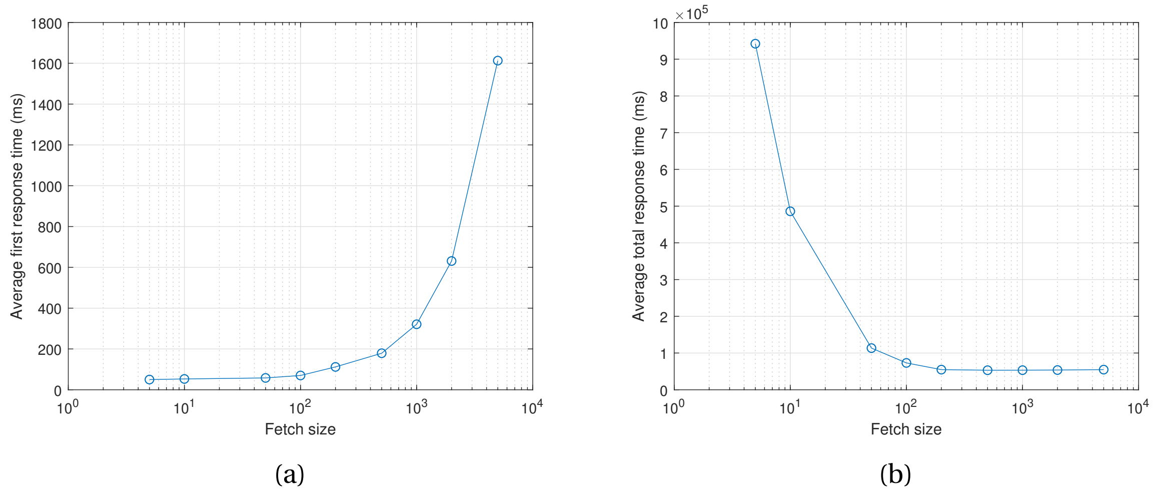
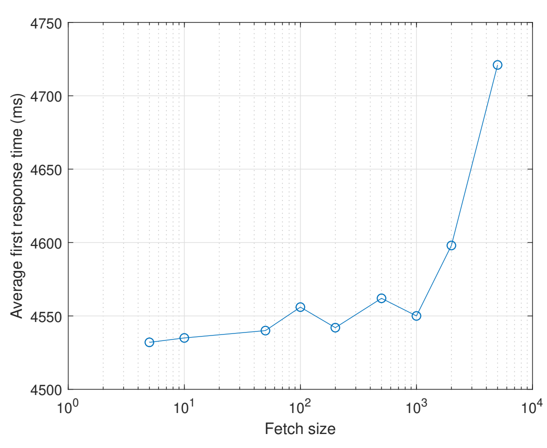
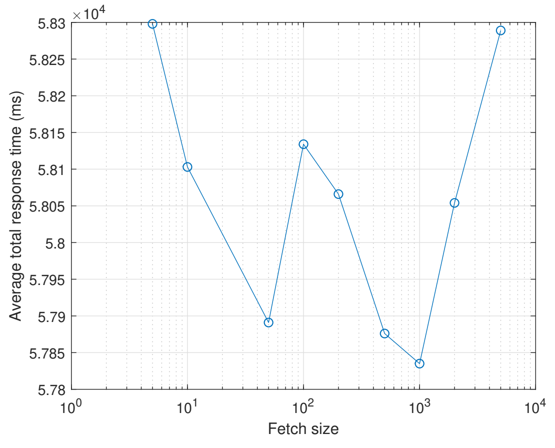


Figure 5.20: The data streaming performance during bounding box filtering with the method of paginating the query.

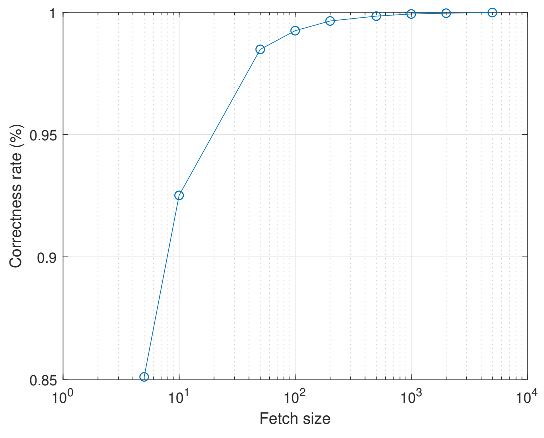
long time before the database starts transmitting data. In terms of total response time, it has a relatively stable mode, about 1 minute, no matter how large the fetch size is. One issue related to this streaming method is that the number of returned CityJSONFeatures is uncontrollable. It is because the fetch size here refers to the number of city objects. When fetching 5 city objects, we may only get one CityJSONFeature when 4 city objects are the children of the other one city object. Another problem is that associated city objects may be not completely queried during one data fetch process. As a result, one CityJSONFeature may be divided into two, resulting more CityJSONFeatures in the end. But the correctness rate increases sharply as the fetch size increases, and approaching to 100% when the fetch size is larger than 100.



(a)



(b)



(c)

Figure 5.21: The data streaming performance during bounding box filtering with the method of streaming the query results

Comparison

The definitions of fetch sizes are different in the two streaming methods. One refers to the number of CityJSONFeatures, and the other refers to the number of city objects. According to the fact that all city objects in *Zurich* dataset are multi-part buildings and the number ratio between city objects and CityJSONFeatures is around 4, we can assume that there is 1 CityJSONFeature when 4 city objects are fetched by the second method.

Based on this assumption, the benchmarking results from the two methods can be displayed together using the same horizontal axis. From the Figure 5.22 we can see that with data streaming from the DBMS to the RESTful API, the first response time is largely reduced while there is not much difference between streaming using the second method and non streaming in terms of total response time. The first streaming method provides much better performance related to receiving the first response, and more acceptable from a user perspective. However, the total response time of the first method is quite long when the fetch size is set to a small value, while the total response time of the second value has a relatively stable mode. Nevertheless, the total response time of the first method improves dramatically as the fetch size increase, and gradually approaching the total response time of the second method. Regarding to the correctness rate, the first method offers total correct result, while the second method depends on the fetch size.

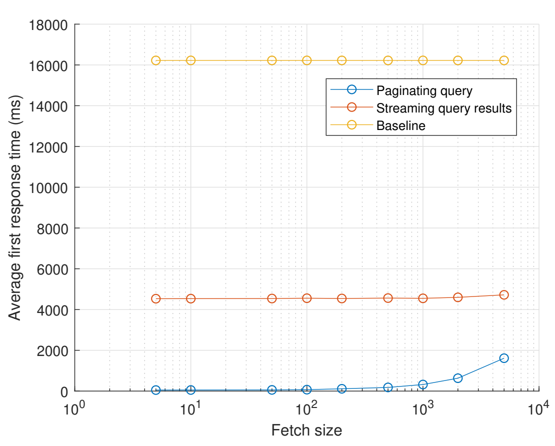
To conclude, the first method is a better option when the fetch size is set between 200 to 2000 in this use case. However, when applying the first method, the optimal fetch size can vary from case to case. The optimal fetch size can be determined based on the total number of the returned CityJSONFeatures, which is unknown before all the data is streamed to the browser. This causes difficulties in applying the first method. on the contrary, the performance of the second method is less affected by the fetch size, so a uniform fetch size can be determined in advance and applied to all use cases. Additionally, the main reason of the poor performance in the second method is that all the city objects in *Zurich* dataset are multi-part buildings. The process of sub-city objects searching and ordering consumes a lot of time. In theory, when the use case is not too complicated, the performance of the second method can be improved.

5.5.3 Use case: attribute filtering on *37en2_volledig* dataset

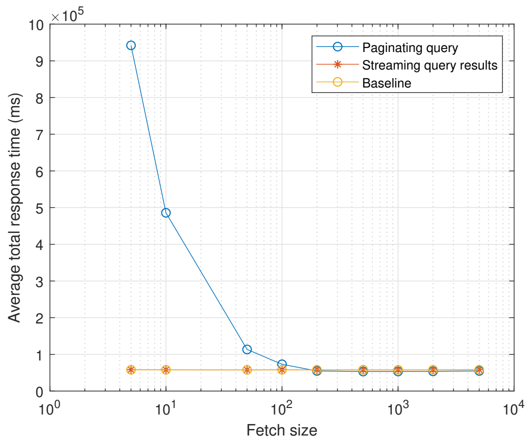
The overview of city objects with different types in *37en2_volledig* dataset is given in Table 5.3. The use case is to find buildings in this dataset, which occupy 19.0% of all the city objects. The HTTP request sent by JMeter is http://127.0.0.1:5000/collections/37en2_volledig/items/?attrs={"type":["Building"]}. Note that all buildings are single-part buildings in *37en2_volledig* dataset, so the correctness rates for both streaming methods stay 100%.

Baseline results

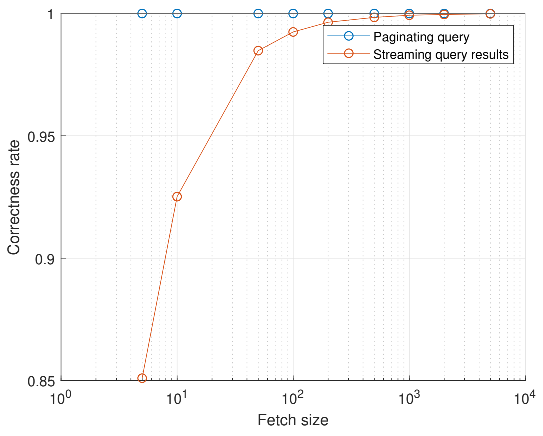
Figure 5.23 shows the baseline results for this use case. From the results we can see that it takes almost half of the total response time for the clients to receive the first CityJSONFeature if the data is not streamed from the DBMS to the RESTful API. It is clear that only streaming data from the RESTful API to the client is not enough for the overall performance. Properly streaming data from the DBMS to the RESTful API is necessary to reduce the first response time, especially when the DBMS is not hosted in the same place (IP) with the RESTful API.



(a)



(b)



(c)

Figure 5.22: The comparison of data streaming performance of bounding box filtering.

Type	Number	Percentage
LandUse	45822	43.1%
Road	20190	19.0%
Building	20053	18.8%
PlantCover	12527	11.8%
WaterBody	4126	3.9%
GenericCityObject	2337	2.2%
Bridge	1362	1.3%

Table 5.3: The overview of city objects with different types in *37en2_volledig* dataset.

Method 1: Paginating the query

From Figure 5.24 we can see that the changing patterns are similar to the first use case. As the fetch size increases, the first response will gradually take longer, but all the first responses are

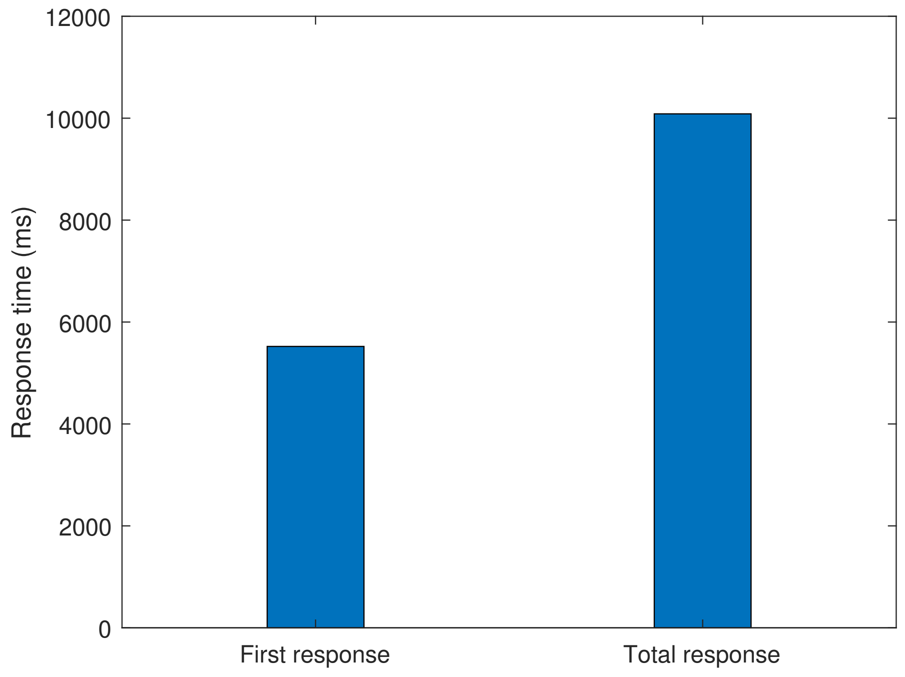


Figure 5.23: The performance during attribute filtering without streaming from the DBMS to the RESTful API .

within 1 second. On the other hand, the total response time decrease dramatically at first, then stays relatively stable. We can also notice that the total response time slightly increases when the fetch size changes from 500 to 5000. This is because the Internet overload occurs due to the limited bandwidth when a large amount of data is fetched from the database at one time.

Method 2: Streaming the query results

Figure 5.25 represents the streaming performance of the second method. The first response time goes up as the fetch size increases, but the difference is less than 400 millisecond. Regardless of fetch size, the total response time has a stable pattern, and slight fluctuates between 9350 and 9800 milliseconds.

Comparison

Figure 5.26 represents the comparison of data streaming performance of attribute filtering between the two methods. Although the first method still offers better performance in first response performance, the difference between the two methods is much reduced when compared with the first use case. This is because the process of querying all results in the database is simpler than in the first case, so it takes less time for the second method to start transmitting data from the DBMS to the RESTful API. On the other hand, the second method has advantages

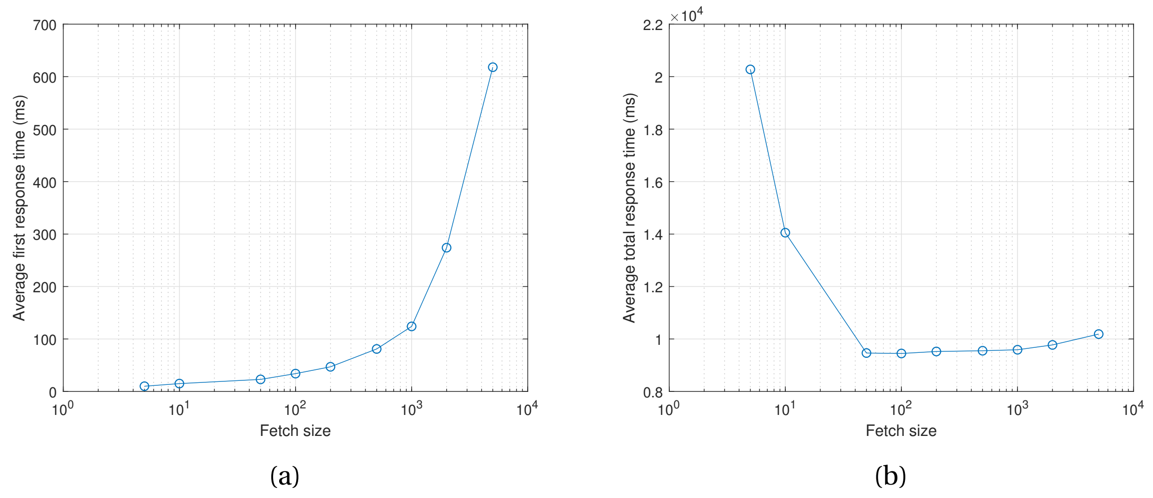


Figure 5.24: The data streaming performance of attribute box filtering with the method of paginating the query.

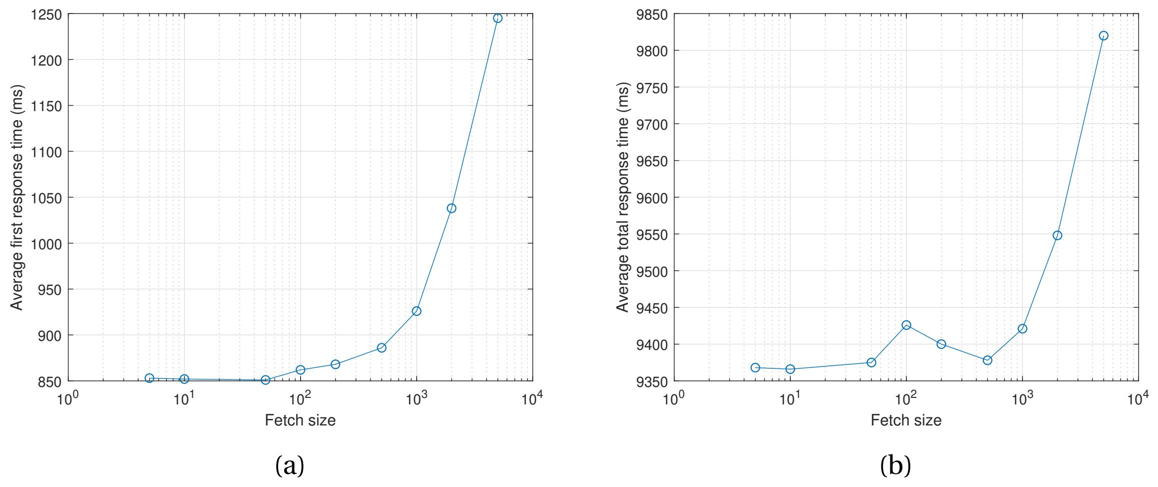


Figure 5.25: The data streaming performance of attribute box filtering with the method of streaming the query results.

in terms of total response time, especially when the fetch size is set to a small value.

Another observation in the second use case is that without data streaming from the DBMS to the RESTful API, the total response time is longer than using the second streaming method. As the fetch size increases from 1000 to 5000, the difference gradually shrinks between not using streaming and using the second streaming method. The possible reason is that when using data streaming, there exist paralleled processes: transmitting the CityJSONFeature to the client and fetching data from the DBMS. However, this speculation can not explain the first use case in terms of the total response time because the difference between not using streaming and using the second streaming method is not significant in the first use case. The possible reason could

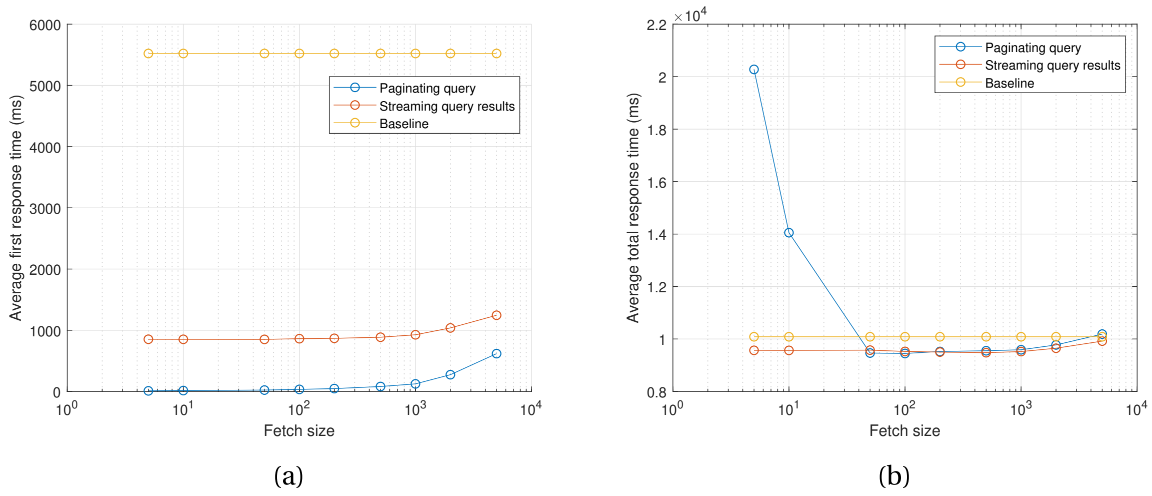


Figure 5.26: The comparison of data streaming performance of attribute filtering between two methods.

be related to the response size. However, more accurate reasoning for this observation can be analysed in the future work.

All in all, the first method is not dominant in this use case. Only when the fetch size is about 50, the first method is better than the second method in terms of the first response time and the total response time. When dealing with different use cases, it is difficult to accurately determine the optimal fetch size for the first method. As a result, streaming performance may not be satisfactory. Therefore, when considering the overall performance and various use cases, the second method can be a better choice to perform data streaming when dealing with less complex CityJSON datasets where most city objects are at the first level. However, one issue with the second method is that one CityJSONFeature may be divided into two CityJSONFeatures. Given the fact that the split only occurs in the two consecutive data fetches from the DBMS, this issue becomes solvable and the detailed solution will be discussed in Section 6.3.

Regarding the first streaming method, the main challenge lies in the determination of the optimal fetch size from case to case. The total number of returned CityJSONfeatures could provide an indication to the optimal fetch size. However, the query in the first streaming method is paginated and the total number of returned CityJSONfeatures can only be known after the streaming process. Even if a total number of the to-be-returned city objects is known, it is still challenging to determine the optimal fetch size because this value not only directly affects the first response time but also has a great impact on the total response time. Because the times of database queries are equal to the total number of the to-be-returned city objects divided by the fetch size. Large times of database queries will lead to poor total response performance. Hence, there is always trade-off between the first response time and the total response time.

Another important observation in the benchmarking results can be used to solve this problem in theory. From Figure 5.20b and Figure 5.24b we can see that the turning point is a more significant feature than a specific optimal point. Because when the fetch size is larger than the turning point, the total response time becomes relatively stable. We can also observe that the acceptable range of fetch size for the first response time is quite large. For example, even if the fetch size is set to 5000, the first response time in the first use case is still less than 2 seconds and less than 700 milliseconds in the second use case.

Based on the above observation, to enable the first method to perform well in various situations, our goal is to find the maximum value of the turning point for the total response time in most use cases. This speculation can be roughly illustrated by the benchmarking results in this research. The values of the turning points are around 200 and 50 separately in the first and second use cases. Hence, the maximum value of the turning points is 200. When we apply 200 as the fetch size in the second use case, the overall performance of the first method is still satisfying (see Figure 5.27). Considering that in the first use case, the total response size is 199.4 MB and the response contains 136418 city objects in total, the value of the turning point in the first use case is sufficient to regard as the maximum value of the turning points in most use cases. Through the theoretical analysis and two benchmarks, a fixed fetch size of 200 can be applied to various use cases when using the first streaming method. However, this speculation still needs to be verified by more use cases with different response sizes. In addition, it is recommended that more work is carried out in the future to find solutions to dynamically determine the fetch size of a specific case.

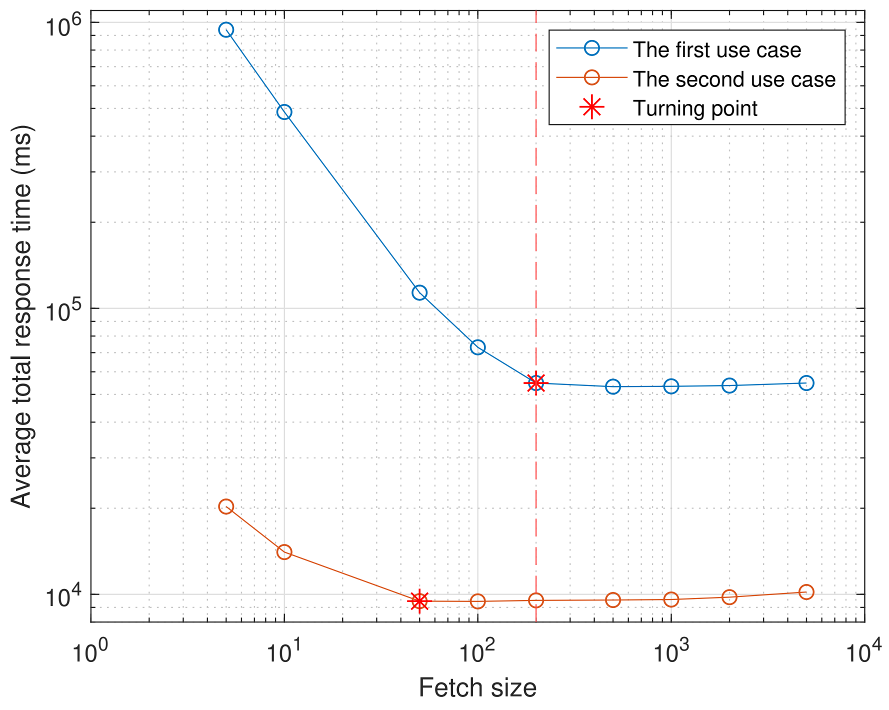


Figure 5.27: Applying the maximum value of the turning points (200) to the second use case.

Chapter 6

Conclusion and future work

In this chapter, a research overview is described and discussed in Section 6.1. The research questions introduced in Section 1.3 are then answered in Section 6.2. In the end, due to some limitations in my methodology, some future work is recommended in Section 6.3.

6.1 Research overview

With the increasing demand for 3D city models in various applications, providing efficient access to 3D city models has become very important and valuable. CityJSON has advantages in data storage and web applications over CityGML data format, thus being chosen as the main datasets used in this research to implement the efficient dissemination of 3D city models on the web.

6.1.1 Developing a RESTful API

Inspired by the *OGC API – Features*, a RESTful API can be developed to disseminate CityJSON datasets with efficiency and scalability. However, there are some divergences between the *OGC API – Features* and my proposed methodology of implementing a RESTful API for CityJSON datasets:

- **Return CRS information**: When the *Collection* resource is accessed, *OGC API – Features* requires to return a list of CRSs used in all the geometries while my methodology returns a simple key/value pair (e.g. "referenceSystem": "urn:ogc:def:crs:EPSG::7415"). This is because all geometries in a CityJSON dataset share the same CRS. Thus, a list of CRSs is replaced by a single string in my methodology.

- **Return spatial extent**: When returning a spatial extent of a *Collection* resource, *OGC API – Features* needs to specify the bounding box and the related CRS (see Listing 2.6). Hence, the spatial extent will be represented by an object with two members, "bbox" and "crs". However, in my proposal, the spatial extent can be simply expressed by an array with 6 elements (e.g. "geographicalExtent": [84710.1, 446846.0, -5.3, 84757.1, 446944.0, 40.9]). This is also the same expression for the spatial extent in a CityJSON object.

- **Return attribute information**: In the *OGC API – Features*, the spatial extent and temporal range of a collection can be optionally contained in the response of accessing the *Collection* resource. However, the other attribute information is neglected. In Section 3.2.4, I propose to include overall information of attributes in the response. The attribute names along with the possible value enumerations or ranges will be sent to the clients when the *Collection* resource is requested.

- **Return queryables** *OGC API – Features* supports to provide queryables by adding the sub directory to the URL for the *Collection* resource: /collections/{collectionId}/queryables. This URL of a GET request will return a list of names that can be used in data filtering process. In my proposal, the queryables are already included in the overall information of attributes. Apart from the attribute names, my proposal also includes the value ranges, which can offer a more comprehensive understanding of the CityJSON dataset to the users. However, the downside of my proposal is that it increases the response size when accessing the *Collection* resource.

- **Handle different CRSs issues**: The *OGC API – Features Part 2* extends the core part's capabilities to use any other CRSs by offering a global list of supported CRS identifiers. A simpler identifier, the EPSG code, is adopted in my methodology to specify the CRS of the filter's bounding box. When returning the CRS information of one *Collection*, the CRS expression is the same with the one stored in CityJSON dataset.

- **Filter on multiple CityJSON datasets**: The core part of *OGC API – Features* only supports filtering operations on a single *Collection* resource. As described in Section 3.2.3, I propose to append two parameters `bbox` and `crs` to the URL path for the *Collections* resource to implement the bounding box filtering on multiple CityJSON datasets. In theory, attribute filtering on multiple CityJSON datasets can also be simply implemented by adding other parameters in the *Collections* URL. However, this use case is less commonly requested than the bounding box filtering on multiple CityJSON datasets, thus not being implemented in my research.

- **Handel filtered results**: The filtered results may contain large amounts of data, which brings the challenge to the RESTful API. The solution in the *OGC API – Features* is to offer paginated filtered results by adding `limit` and `offset` parameters in the HTTP request path. This means the users need to proactively browser the next 10 filtered features (if the `limit` is set to 10) by sending a new request to the RESTful API again. My solution is to stream all the filtered data. In this way, the user only needs to send the HTTP request to the RESTful API once and then the browser will incrementally receive and parse all the filtered results. Data streaming allows the user to view the overall filtered results on one page instead of the divided results on multiple pages. However, one downside about data streaming is that the browser needs to accommodate all the data on one page at the end of the streaming process, which may exceed its capacity and cause the browser to crash.

- **Construct filtering expressions**: The draft version of *OGC API – Features Part 3* introduces a Common Query Language for constructing filtering expressions. Each filter expression consists of a key/value pair where the key refers to the operator (see Listing 2.8). However, in my proposal, the key refers to the attribute name (see Listing 3.2).

  The draft version of *OGC API – Features Part 3* provides comprehensive guidance for constructing filtering expressions. Due to the limited time and the unavailability of the official *OGC API – Features Part 3* document, only the most commonly used operations are implemented in my own methodology (see Table 6.1). The intersection is the only spatial operation in my research because this is meant for the bounding box filtering process. Additionally, the filtering expressions used in my work do not follow the Common Query Language. More future work can be added to enrich the filtering capabilities in my RESTful API with a consistent rule of filtering expressions.

Class	**Operator**	***OGC API – Features***	**Own methodology**
Comparison	equal to	✓	✓
	greater than	✓	✓
	less than	✓	✓
	greater than or equal to	✓	✓
	less than or equal to	✓	✓
	between	✓	✓
	like	✓	✗
	is NULL	✓	✗
	in	✓	✓
Logical	and	✓	✓
	or	✓	✗
	not	✓	✗
Spatial	intersects	✓	✓
	equals	✓	✗
	... (6 more)	✓	✗
Temporal	anyinteracts	✓	✗
	... (14 more)	✓	✗

Table 6.1: The set of operators considered in *OGC API – Features* and my own methodology

6.1.2 Data streaming

CityJSONFeature enables CityJSON data to be parsed incrementally on the web. Another issue that may cause high latency for users to receive the first CityJSONFeature is the data processing and transmission from the DBMS to the RESTful API. To improve the overall data streaming performance, I proposed two methods in Section 3.2.5 to stream the data from the DBMS to the RESTful API so that the RESTful API can start constructing the first CityJSONFeature and sending it to the user without having to wait for all data tuples to be transferred from DBMS. The most challenging part of data streaming from the DBMS to the RESTful API is to correctly bundle together the city objects at the second level with their parent city objects. My solution is to append an extra column that indicates the main id in the query results for the RESTful API to differentiate the features, and then sort the query results by the main id so that the RESTful API can construct the CityJSONFeatures in a streamed way.

The benchmarking results show that if the data is not streamed from the DBMS to the RESTful API, it takes around 16 seconds for the user to receive the first CityJSONFeature in the first use case and takes almost half of the total response time in the second use case. Both of these streaming methods have shown to largely reduce the first response time. However, there are some performance differences between the two methods:

- **First response**: The first method performs better than the second method, especially when the query is complex and result is large. In the first use case for benchmarking the data streaming performance, users will receive the first CityJSONFeature after 4.5 second when the second method is used. This is because the *Zurich* dataset only contains the multi-part building and the filtered result (about 200 MB) accounts for 85% of the *Zurich* dataset. When the query process is less complex, for example, in the second use case, the performance of the second method becomes much acceptable (less than 1 second) and close to the first method.

  Theoretically speaking, although the second method does not perform well in terms of the first response, it can provide better performance in the rest of responses, such as the second one. This is because there is no need to query database again in the second method. The speculation can also be proven by the mathematical principles since the second method provides less (or same) total response time than the first method. However, this speculation still needs future work to verify.

- **Total response**: As discussed above, the second method has advantages in the total response in comparison to the first method. This is because the database query is a time-consuming and resource-intensive process. When the fetch size is set to a small value, there may be thousands of database queries in the first method to retrieve all the data. Generally, the second method is less sensitive to the fetch size while a proper fetch size should be determined to avoid a long total response time in the first method. On the other hand, the fetch size can not be set to a very large value (e.g. 10000) since it will dramatically increase the first response time in both methods.

- **Correctness**: In the first method, the fetch size directly limit the number of city object at the top level, and then the associated city objects are queried and sent along with their parent city objects to the RESTful API. However, in the second method, the fetch size refers to the total number of city objects the RESTful API will retrieve at one time. Although the sort operation in the SQL ensures that the city objects at the second level are bundled together with their parent city objects, city objects contained in a CityJSONFeature may not be fetched together at one time in the second method. This will cause the split of the CityJSONFeature.

As mentioned in Section 5.5.3, considering that the split only occurs in the two consecutive data fetches from the DBMS in the second method, this issue becomes solvable without breaking the data streaming rule, and the detailed proposal for fixing this issue will be discussed in Section 6.3.

- **Adaptability to various use cases**: A normal fetch size like 10, 100 and 500 can be applied in the second method for various use cases. However, the fetch size should be carefully picked for the first streaming method because inappropriate fetch sizes will lead to a very long total response time. The total number of returned city objects could indicate the optimal fetch size but this value is unknown until the entire data streaming process is finished in the first method.

  As discussed in Section 5.5.3, the turning point of the total response time is a significant feature and can be used to replace the optimal point. Based on the theoretical speculation, the research goal becomes to find the maximum value of the turning point values among most use cases. Considering that the first use case is already much more complex and challenging than normal use cases, its value of the turning point (200) is large enough to be the fixed fetch size for various use cases. However, more use cases need to be benchmarked to verify this speculation. In addition, it is recommended that more work is carried out in the future to find solutions to dynamically determine the fetch size of a specific case when using the first streaming method.

When applying my proposed streaming methods to CityJSON-related web applications, the first streaming method with a fixed fetch size of 200 is recommended since it offers a fast first response time, acceptable total response time, and entirely correct results. If the requirement of bundling child city objects together with their parent city objects is removed from the web application, the second streaming method is better than the first one because the SQL can be largely simplified and the time-consuming sort operation is not needed anymore. In addition, the fixed fetch size of 200 for the first method has not been systematically verified yet and there is no solution proposed for the the first method to dynamically determine the fetch size on a case-by-case basis before the data streaming process starts.

6.1.3 Storing CityJSON in a database

Despite the fact that file-based systems are the easiest way of data storage, they have deficiencies in some applications that require scalability, efficiency and flexibility in data access. Therefore, a database solution for storing CityJSON is studied in my research. The drawbacks of storing CityJSON using 3DCityDB are discussed in Section 2.4.2, so 3DCityDB is not adopted to store CityJSON and a new database solution is designed and implemented in *PostgreSQL*.

DBMS vs file system

To comprehensively compare the performance between the DBMS and file system, 5 use cases in the RESTful API are designed and the web performance is discussed in Section 5.4. A summarized benchmarking result is given in Table 6.2.

Use case	**Description**	**Preferred**
Access to one specific city object	When the dataset size reaches **100 MB**, it takes around **20 seconds** for the file system to response while the response time for the DBMS is around **10 milliseconds** even with a dataset size of **450 MB**.	DBMS
Access to 10 random city objects	When a dataset with size of **450 MB** is accessed, the file system needs **6 minutes** while the response time with the DBMS is constantly below **100 milliseconds**	DBMS
Access to a whole CityJSON dataset	The file system constantly responses faster than the DBMS and the difference start enlarging as the dataset size reaches **100 MB**. The difference is **1.70 times** with a size of **600 MB** and **1.38 times** with a size of **286 MB**, respectively.	File-based system
Bounding box filtering	When the filtering range is small, the DBMS performs much better than the file system, and the advantage will be weakened as the range increases. For example, when there are only **0.1%** city objects in the filtering range, the time taken by the file system is **85 times** that of DBMS. When the range reaches **50%**, the time is only **1.25 times** that of DBMS.	DBMS
Attribute filtering	Similar pattern with the bounding box filtering	DBMS

Table 6.2: A summarized benchmarking result for the storage system.

From the summarized results we can conclude that with the help of the built-in query and index mechanism, the DBMS performs better than the file system in most use cases. However, when a whole CityJSON dataset is requested, the file system has a faster response than the DBMS. Therefore, a hybrid storage management solution integrating the file-based system and DBMS is recommended to support the fast access in the RESTful API.

Database schema design

In the database schema proposed by Staring [2020], city objects are fragmented into 3D surfaces, which is not necessary in my research. By following the *OGC API – Features*, I defined the minimum accessible resource as a city object. This means that the data access and filtering process in my research scope only works at the city object level. Additionally, it is difficult and time-consuming to reconstruct the city object from 3D surfaces, and the reconstruction process will lead to a performance penalty in the RESTful API. To better support the RESTful API, I proposed a different database schema, where there are only two tables *CityJSON* and *Cityobject*. *Cityobject* table stores each city object in a row with a local list of vertices to support the efficient access to the *Feature* resource and data streaming. JSONB data type is used in many columns to give a compact database schema that includes flexible storage.

To support data streaming and filtering in the RESTful API, the type value, 2D bounding box and parent_child relation are derived from the CityJSON dataset and explicitly added into database schema. Different ways of storage for the three parts are proposed in Section 3.3. To search for the best way to support the RESTful API, the advantages and disadvantages of the different ways of storing the type value (2D bounding box and parent_child relation) are summarized according to the benchmarks and analysis in Section 5.3:

- **Semantics (type)**: According to the Table 6.3, adding an extra *text* column is better than storing the type value in the *JSONB* column along with other attributes. Because the traditional *text* data type has better query planner and there is no big difference in the overall storage occupation compared to the *JSONB* data type.

Method	Pros	Cons
Text	1) better query performance on normal use cases (3% faster); 2) better query planner; 3) much faster query performance on special cases (1.7 times faster)	1) higher storage occupation for pure data (6.7% more); 2) more overall storage (0.8% more but neglectable
JSONB	1) less storage occupation for pure data; 2) more compact schema by reducing one column in *Citybject* table	1) worse query performance; 2) more complicated indexing mechanism; 3) lacking statistics data for better query planner; 4) extra space needed (10% more)

Table 6.3: Summarized pros-and-cons analysis on type value storage

- **2D bounding box**: Based on Table 6.4, *Box* type has the lowest overall storage size and significantly better query performance, thus being chosen to store 2D bounding boxes.
- **Parent-child relation**: From Table A.1 we can conclude adding a parent_id column with *text* type is the best solution. Because it has acceptable query performance, does not cause severe redundancy issue and provides flexibility in searching parent ID from the child city objects.

Based on the benchmarks and analysis, the optimal database schema for supporting the RESTful API is given in Figure 6.1. Although the color-marked columns will increase the database storage, they can help extend the capabilities of the RESTful API with data streaming, bounding box filtering, and attribute filtering.

Apart from the above research results, my contribution also lies in the implementation of storing CityJSON in a database with spatial coherence by SFC indexing (Section 4.2.2) and the

Method	**Pros**	**Cons**
Geometry	1) related CRS storage supported; 2) more built-in spatial function (ST_Transform, &&); 3) spatial index support 4) 3D supported	1) worse query performance; 2) complex data expression
Box	1) built-in spatial function supported (&&); 2) lowest overall storage size; 3) best query performance (2 times faster)	1) no CRS support; 2) not 3D extensible
Array	1) simple value expression; 2) 3D extensible	1) worse query performance; 2) no CRS support; 3) no built-in spatial function and index

Table 6.4: Summarized pros-and-cons analysis on 2D bounding box storage.

Method	**Pros**	**Cons**
Array & JSONB (Array)	1) easy query of child IDs	1) bad query performance; 2) difficult in searching parent ID from child city objects (single-direction of searching)
Text	1) second best query performance; 2) flexible in data query (double-direction of searching)	
Feature (JSONB)	1) best query performance	1) severe redundancy with *Zurich* dataset; 2) single-direction of searching

Table 6.5: Summarized pros-and-cons analysis on parent-child relation storage.

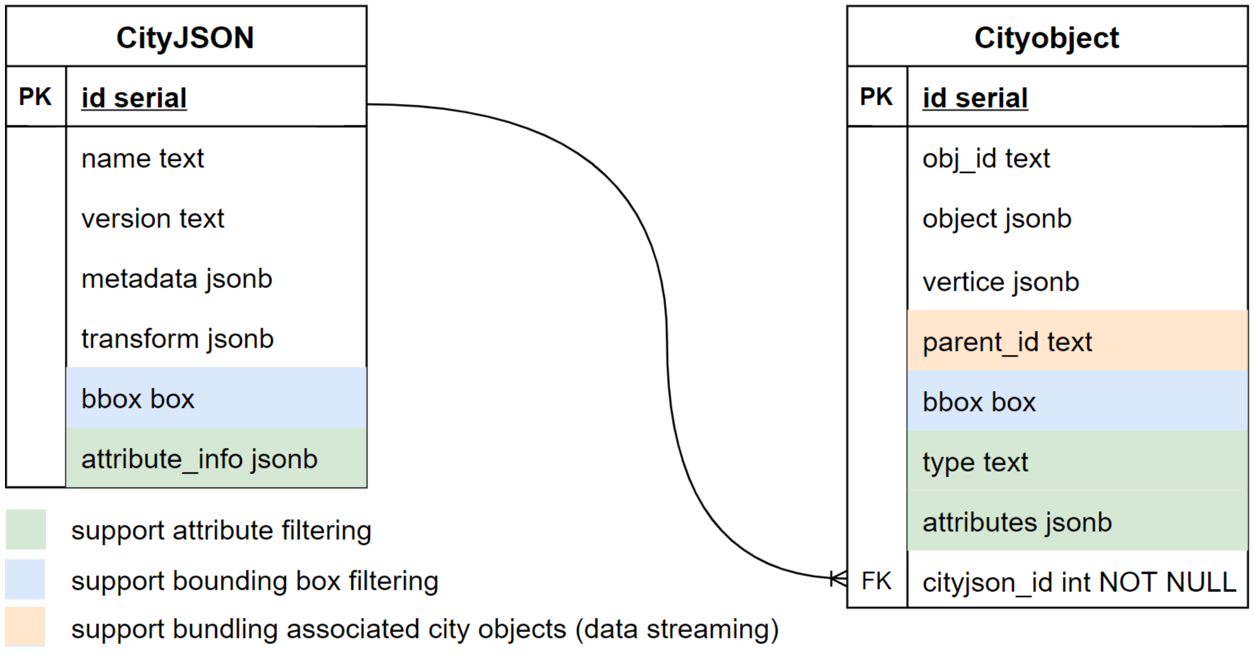


Figure 6.1: The final database schema for storing CityJSON.

design of a dynamic normalization method to support the streaming with visualization (Section 4.2.5).

6.2 Research question

Main question: How to best develop a RESTful API for fast access to geospatial features in CityJSON and how to properly store CityJSON in a database to support the RESTful access? is addressed by giving answers to the sub-questions below and combining those answers with the the discussion and conclusion in Section 6.1.

Sub question 1: How to mapping the data in CityJSON datasets to the resources defined by *OGC API – Feature*?

- The resource mapping is discussed in Section 3.2.1. Each city object is mapped to a *Feature* resource. *Collection* resource only contains the higher level properties such as the CRS and spatial extent.
- The child city objects will be bundled together with their parent city objects when their parent city objects are accessed. However, a child city object can be independently send to the client as a *Feature* if requested.
- Some divergences between the *OGC API – Features* and my proposed methodology are made to better work with CityJSON datasets and extend the capabilities of the RESTful API. These divergences are discussed and concluded in Section 6.1.1.

Sub question 2: How to solve the CRS issues during data access?

- The CRS for each CityJSON dataset is store in *CityJSON* table. Thus, the CRS for each city object can refer to the associated *CityJSON* by the foreign key *cityjson_id*.
- To avoid the ambiguous CRS during the bounding box filtering process, users are required to provide the EPSG code along with the 2D bounding box to the RESTful API. Considering that *Box* data type is chosen to store 2D bounding boxes, `ST_Transform` is not applicable for this data type in *PostgreSQL*. Hence, a CRS conversion in the RESTful API is needed when the filter's CRS is different to the 2D bounding boxes in the database.

Sub question 3: How to implement the efficient querying of city objects based on geometries and semantics?

- To support the efficient data filtering on city objects, 2D bounding boxes, type, attributes are respectively stored in *Box*, *text* and *JSONB* columns in *Cityobject* table.

- The geometric filtering is simplified to the bounding box filtering. The spatial intersect operation `&&` and spatial indexing mechanism in *PostgreSQL* can work on the *Box* column. Additionally, the indexing mechanism can also be applied on the JSONB column.

Sub question 4: How to define streaming in the geospatial context and how to stream CityJSON datasets?

- Data streaming in the geospatial context means that the browser can parse and proceed geospatial data incrementally without having to receive the entire dataset.
- In order to implement data streaming on CityJSON dataset, the city objects are modified to CityJSONFeatures so that each city object can be processed independently.
- Data streaming also involves the data transmission process from the DBMS to the RESTful API. I proposed two methods to handle this issue and the performance differences between the two methods are concluded in Section 6.1.2.

Sub question 5: Which database is suitable for storage of CityJSON datasets to support RESTful access and how to design the database schema?

- As discussed in Section 2.4.2, due to the availability of spatial data types, functions and indexing mechanisms, a spatially-extended relational database is chosen in this research, more specifically *PostgreSQL* with *PostGIS* extension.
- The database schema design is discussed and concluded in Section 6.1.3.

6.3 Future work

Due to some limitations in my methodology, several future work ideas to further improve the API are discussed and recommended in this section:

- **Data compression in database systems**: The network bandwidth sets a bottleneck to the efficiency of the data transmission on the web. To find the best method to compress CityJSON files, Van Liempt [2020] compares the performance improvement using different compression methods and concludes that CBOR, zlib, or a combination of these two can best compress CityJSON files. This research result can be integrated to the RESTful API for better efficiency when a whole CityJSON file is requested.

  In my research, a database solution has been proposed for the storage of CityJSON datasets. Compression on the database can also contribute to the overall performance. Some preliminary compression work on PostgreSQL has already been investigated by some scholars and organizations. For instance, Postgres 9.5 released a new setting called "wal_compression", which is able to reduce the IO load [Moppel, 2016]. In addition, Aya [2019] introduced the columnar compression in Zedstore. The author's benchmarks showed a good improvement in terms of loading time and disk footprint. Those research results and methodologies can be implemented in my model to further improve the database performance.

- **Eliminating the split of CityJSONFeature in the second streaming method**: One criticism of the second streaming method is that the split of CityJSONFeature could happen when the associated city objects are not fetched at the same time. However, there are two useful characteristics in the split. First is that the split only occurs in the two consecutive data fetches from the DBMS and the other one is that the parent city object will always be in the first split part. Based on those two characteristics, this issue becomes solvable and my solution is suggested as follows.

  The last constructed CityJSONFeature during one data fetch can be temporarily stored in the memory using a variable. When the new city objects are fetched to the RESTful API, a comparison can be made to check whether the first city object in the new data fetch is a parent city object. This is done by comparing the main id and the the city object id. If the main id is the same with the the city object id, it can be determined that the city object is a parent. Hence, if the first city object in the new data fetch is a parent city object, this means that the temporarily stored CityJSONFeature contains all the associated child city objects and can be sent to the client immediately. If not, the temporarily stored CityJSONFeature needs to add some child city objects from the new data fetch until the first parent city object in the new data fetch shows up.

- **Determining optimal fetch size in the first streaming method**: The observation and theoretical analysis in Section 5.5 suggests that the maximum value of the turning point for the total response time in most use cases is applicable to a random use case. This speculation needs to be verified by more use cases with different total number of returned CityJSONFeatures.

  It is also recommended that more work should be carried out to find solutions to dynamically determine the fetch size on a case-by-case basis before the data streaming process starts. Theoretically, the total number of returned CityJSONfeatures is a great indication of the optimal fetch size. However, this value in the first method can only be known after the streaming process. To fix this issue, I propose to have an additional query before the real data retrieval process. The additional query is meant to get the count value from the database by the simple SQL syntax `COUNT(*)`. Although this will unavoidably increase the first response time, the count operation in the database does not need to inspect the actual data in rows and is supposed to be much faster than normal query. However, this speculation needs to be benchmarked.

  There is always trade-off between the first response time and the total response time. I propose to add more experiments to find the maximum fetch size boundary for the first response time and the maximum total fetch times for the total response time. Let $Total_{Maxfetch}$ and $Size_{Maxfetch}$ be the boundary values for the maximum total fetch times and maximum fetch size. Once the total number of the to-be-returned CityJSON-Features ($Total_{CityObject}$) is determined, the reasonable fetch size ($Size_{fetch}$) can be decided by the equation below (see Equation 6.1).

$$\frac{Total_{CityObject}}{Total_{Maxfetch}} < Size_{fetch} < Size_{Maxfetch} \tag{6.1}$$

- **Extending 2D bounding box to 3D**: Currently, the bounding box filtering only works in 2D. To extend to 3D, the recommended *Box* type is not applicable anymore. Although *Geometry* type supports the 3D objects, it can only store a 3D surface instead of a 3D bounding box (solid). To address this problem, the 3D bounding box can be simplified to a 3D surface (see Figure 6.2). Another option is to explicitly store each 3D surfaces in CityJSON datasets. In this case, the bounding box filtering can directly work on the those 3D surfaces without the auxiliary bounding box column.

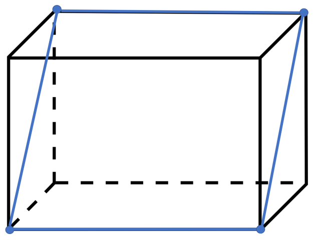


Figure 6.2: The 3D bounding box represented by a 3D surface (bounded by blue lines).

- **Extending the minimum accessible resource**: Since the database schema I proposed has been specifically designed for supporting RESTful access to geofeatures in CityJSON datasets, this schema is not applicable to other applications that require more detailed resources in CityJSON such as LoD. However, the database schema is extendable to meet the new requirements by adding extra *Geometry* table or columns.

- **Complying to the CQL**: Due to the time limitation, only the most commonly used operations are implemented in my research. Since *OGC API – Features Part 3* provides comprehensive guidance for constructing filtering expressions, the filtering expressions in my research can be modified by following the unified CQL rule and more operations can be added.

- **Fixing the issue of having too large single city objects**: In a CityJSON dataset, there may exist vary large city objects (e.g. Land use). Although only one CityJSONFeature is sent to the browser, the data streaming performance is still degraded by a very large city object. More future work is recommended to stream data at lower level (e.g. geometry) or some pre-process work can be done to fix this issue.

Bibliography

Nosql based 3D city model management system. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, 40(4):169–173, 2014. ISSN 16821750. doi: 10.5194/isprsarchives-XL-4-169-2014.

3DCityDB Team. Welcome to 3D City Database documentation! — 3dcitydb-docs 4.2.3 documentation, 2019. URL `https://3dcitydb-docs.readthedocs.io/en/release-v4.2.3/`.

Apache JMeter™. Apache JMeter - Apache JMeter™, 2021. URL `https://jmeter.apache.org/`.

I. Aya. Data compression in PostgreSQL and its future. Technical report, 2019.

D. Baston. Storage-Optimizing PostGIS Geometries, 2018. URL `http://www.danbaston.com/posts/2018/02/15/optimizing-postgis-geometries.html`.

F. Biljecki, J. Stoter, H. Ledoux, S. Zlatanova, and A. Çöltekin. Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information*, 4:2842–2889, 2015.

F. Biljecki, K. Kumar, and C. Nagel. CityGML Application Domain Extension (ADE): overview of developments. *Open Geospatial Data, Software and Standards*, 3(1):1–17, 2018. ISSN 2363-7501. doi: 10.1186/s40965-018-0055-6.

R. Billen, A.-F. Cutting-Decelle, O. Marina, J.-P. de Almeida, C. M., G. Falquet, T. Leduc, C. Métral, G. Moreau, J. Perret, G. Rabin, R. San Jose, I. Yatskiv, and S. Zlatanova. 3D City Models and urban information: Current issues and perspectives. pages I–118. EDP Sciences, 2014. doi: 10.1051/tu0801/201400001. URL `www.edpsciences.orgwww.webofconferences.org`.

C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON Document Stores in Relational Systems (not published long version). *Proceedings of the 16th International Workshop on the Web and Databases 2013 (WebDB 2013)*, pages 1–6, 2013. URL `https://www.microsoft.com/en-us/research/publication/enabling-json-document-stores-in-relational-systems/`.

CityJSON. Datasets | CityJSON, 2021. URL `https://www.cityjson.org/datasets/`.

E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, jun 1970. ISSN 15577317. doi: 10.1145/362384.362685. URL `https://dl.acm.org/doi/10.1145/362384.362685`.

K. D. Foote. A Brief History of Non-Relational Databases - DATAVERSITY, 2018. URL https://www.dataversity.net/a-brief-history-of-non-relational-databases/.

M. Drake. A Comparison of NoSQL Database Management Systems and Models | DigitalOcean, 2019. URL https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models.

B. Dukai, R. Peters, T. Wu, T. Commandeur, H. Ledoux, T. Baving, M. Post, V. Van Altena, W. Van Hinsbergh, and J. Stoter. GENERATING, STORING, UPDATING and DISSEMINATING A COUNTRYWIDE 3D MODEL. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, 44:27–32, 2020.

T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian. *SOA with REST: Principles, Patterns Constraints for Building Enterprise Solutions with REST*. 08 2012. ISBN 9780137012510.

J. Freeman. What is JSON? A better format for data exchange | InfoWorld, 2019. URL https://www.infoworld.com/article/3222851/what-is-json-a-better-format-for-data-exchange.html.

G. Gröger, T. H. Kolbe, C. Nagel, and K.-H. Häfele. OpenGIS City Geography Markup Language (CityGML) Encoding Standard, Version 2.0.0. *OGC Document No. 12-019*, page 344, 2012. URL https://portal.opengeospatial.org/files/?artifact_id=47842.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15:287–317, 1983. ISSN 15577341. doi: 10.1145/289.291. URL https://dl.acm.org/doi/10.1145/289.291.

D. Hilbert. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

C. Holmes. WFS 3.0 — Get Excited? Yes!. Well, get excited if you're deep in to… | by Chris Holmes | Medium, 2017. URL https://cholmes.medium.com/wfs-3-0-get-excited-yes-8e904fdbcc0.

ISO. ISO - ISO/IEC 9075-1:2016 - Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework), 2016. URL https://www.iso.org/standard/63555.html.

G. Jansen. Resources, 2011. URL https://restful-api-design.readthedocs.io/en/latest/resources.html.

D. Kirkpatrick. Google: 53% of mobile users abandon sites that take over 3 seconds to load | Marketing Dive, 2016. URL https://www.marketingdive.com/news/google-53-of-mobile-users-abandon-sites-that-take-over-3-seconds-to-load/426070/.

T. H. Kolbe. Representing and exchanging 3D city models with CityGML. *Lecture Notes in Geoinformation and Cartography*, (January 2009):15–31, 2009. ISSN 18632351. doi: 10.1007/978-3-540-87395-2_2.

J. K. Lawder and P. King. Using space-filling curves for multi-dimensional indexing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1832, pages 20–35. Springer Verlag, 2000. ISBN 3540677437. doi: 10.1007/3-540-45033-5_3. URL https://link.springer.com/chapter/10.1007/3-540-45033-5_3.

H. Ledoux. cityjson_ogcapi/best-practice.md, 2020. URL https://github.com/hugoledoux/cityjson{_}ogcapi/blob/master/best-practice.md.

H. Ledoux, K. Arroyo Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis. CityJSON: A compact and easy-to-use encoding of the CityGML data model, 2019.

Z. H. Liu, B. Hammerschmidt, D. McMahon, Y. Liu, and H. J. Chang. Closing the functional and Performance Gap between SQL and NoSQL. pages 227–238. Association for Computing Machinery (ACM), jun 2016. doi: 10.1145/2882903.2903731.

J. Long. ndjson, 2021. URL http://ndjson.org/.

C. Lück. Introducing streaming newline-delimited JSON (NDJSON) with ReactPHP, 2018. URL https://clue.engineering/2018/introducing-reactphp-ndjson.

W. McKinney. *Python for Data Analysis*. O'Reilly Media, Inc., 2017. ISBN 9781491957660. URL https://www.oreilly.com/library/view/python-for-data/9781449323592/.

ModernSQL. What's New in SQL:2016, 2017. URL https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016.

K. Moppel. WAL compression - PostgreSQL underused features - CYBERTEC, 2016. URL https://www.cybertec-postgresql.com/en/postgresql-underused-features-wal-compression/.

G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

NoSQL. NoSQL Databases List by Hosting Data - Updated 2021, 2021. URL https://hostingdata.co.uk/nosql-database/.

OGC. Simple Feature Access - Part 1: Common Architecture | OGC, 2021. URL https://www.ogc.org/standards/sfa.

L. Olivera. Everything you need to know about NoSQL databases - DEV Community, 2019. URL https://dev.to/lmolivera/everything-you-need-to-know-about-nosql-databases-3o3h.

Open Geospatial Consortium. CityGML UML diagrams Version 2.0. (12):2002–2012, 2012. URL http://www.citygml.org/fileadmin/citygml/docs/CityGML_2_0_0_UML_diagrams.pdf.

Open Geospatial Consortium. OGC API - Features - Part 1: Core, 2019. URL http://docs.ogc.org/is/17-069r3/17-069r3.html.

Open Geospatial Consortium. An Introduction to OGC API - Features — OGC e-Learning 2.0.0 documentation, 2020. URL http://opengeospatial.github.io/e-learning/ogcapi-features/text/basic-main.html.

J. A. Orenstein and F. A. Manola. Probe Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Transactions on Software Engineering*, 14(5), 1987. ISSN 00985589.

PDOK. PDOK - 3D downloads, 2021. URL https://3d.kadaster.nl/basisvoorziening-3d/.

G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, mar 1890. ISSN 00255831. doi: 10.1007/BF01199438. URL https://link.springer.com/article/10.1007/BF01199438.

I. Pispidikis and E. Dimopoulou. Citygml restful web service: automatic retrieval of citygml data based on their semantics. principles, guidelines and bldg conceptual design. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 4:49–56, 2018.

R. Polding. Databases: Evolution and Change, 2018. URL https://medium.com/@rpolding/databases-evolution-and-change-29b8abe9df3e.

PostgreSQL. PostgreSQL: Documentation: 9.1: Statistics Used by the Planner, 2021a. URL https://www.postgresql.org/docs/9.1/planner-stats.html.

PostgreSQL. PostgreSQL: Documentation: 13: 8.14. JSON Types, 2021b. URL https://www.postgresql.org/docs/current/datatype-json.html.

PostgreSQL. PostgreSQL: Documentation: 9.4: System Administration Functions, 2021c. URL https://www.postgresql.org/docs/9.4/functions-admin.html.

S. Psomadaki. Using a Space Filling Curve for the Management of Dynamic Point Cloud Data in a Relational DBMS. Technical report, 2016. URL https://repository.tudelft.nl/islandora/object/uuid%3Ac1e625b0-0a74-48b5-b748-6968e7f83e2b.

M. Richards. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015. ISBN 9781491925409.

D. Robinson. Heap: When To Avoid JSONB In A PostgreSQL Schema, 2016. URL https://heap.io/blog/engineering/when-to-avoid-jsonb-in-a-postgresql-schema.

S. Gillies. RFC 8142 - GeoJSON Text Sequences, 2017. URL https://tools.ietf.org/html/rfc8142.

A. Stadler, C. Nagel, G. König, and T. H. Kolbe. Making interoperability persistent: A 3D geo database based on CityGML. In *Lecture Notes in Geoinformation and Cartography*, pages 175–192. Kluwer Academic Publishers, 2009. ISBN 9783540873945. doi: 10.1007/978-3-540-87395-2_11. URL https://link.springer.com/chapter/10.1007/978-3-540-87395-2_11.

K. Staring. Combination of CityJSON with PostgreSQL, MongoDB and GraphQL. Technical report, 2020. URL https://repository.tudelft.nl/islandora/object/uuid{%}3A7f9209f7-248f-4c93-9ba3-5866655e6040.

R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures, 2000. URL https://www.researchgate.net/publication/216797523_Architectural_Styles_and_the_Design_of_Network-based_Software_Architectures.

L. Van Den Brink, P. Barnaghi, J. Tandy, G. Atemezing, R. Atkinson, B. Cochrane, Y. Fathy, R. García Castro, A. Haller, A. Harth, K. Janowicz, Kolozali, B. Van Leeuwen, M. Lefrançois, J. Lieberman, A. Perego, D. Le-Phuoc, B. Roberts, K. Taylor, and R. Troncy. Best practices for publishing, retrieving, and using spatial data on the web, 2018.

J. Van Liempt. CityJSON: does (file) size matter? Technical report, 2020. URL https://repository.tudelft.nl/islandora/object/uuid%3A4aad07f4-8f64-46b1-aad3-3d4abe36c5bf.

W3C Working Group. Spatial Data on the Web Best Practices, 2017. URL https://www.w3.org/TR/sdw-bp/.

J. Wendel, A. Simons, A. Nichersu, and S. M. Murshed. Rapid development of semantic 3D city models for urban energy analysis based on free and open data sources and software. In *Proceedings of the 3rd ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics, UrbanGIS 2017*, volume 2017-January, New York, NY, USA, 2017. ACM. ISBN 9781450354950. doi: 10.1145/3152178.3152193. URL https://doi.org/10.1145/3152178.3152193.

M. Winand. Effects of ORDER BY and GROUP BY on SQL performance, 2021. URL https://use-the-index-luke.com/sql/sorting-grouping.

Z. Yao, C. Nagel, F. Kunde, G. Hudra, P. Willkomm, A. Donaubauer, T. Adolphi, and T. H. Kolbe. 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, 3(1):5, dec 2018. ISSN 2363-7501. doi: 10.1186/s40965-018-0046-7. URL https://opengeospatialdata.springeropen.com/articles/10.1186/s40965-018-0046-7.

X. Zhou, X. Wang, Y. Zhou, Q. Lin, J. Zhao, and X. Meng. RSIMS: Large-Scale Heterogeneous Remote Sensing Images Management System. *Remote Sensing*, 13(9):1815, 2021. ISSN 20724292. doi: 10.3390/rs13091815.

Appendices

Appendix A

Reproducibility self-assessment

A.1 Marks for each of the criteria

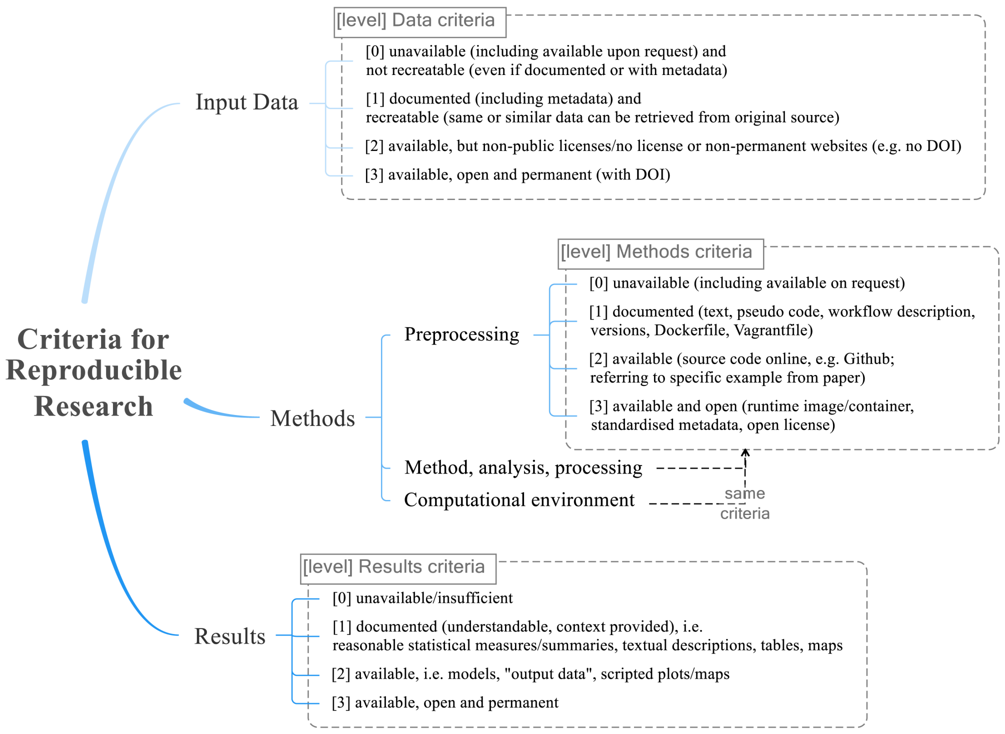

Criteria for Reproducible Research
Input Data
[level] Data criteria
[0] unavailable (including available upon request) and not recreatable (even if documented or with metadata)
[1] documented (including metadata) and recreatable (same or similar data can be retrieved from original source)
[2] available, but non-public licenses/no license or non-permanent websites (e.g. no DOI)
[3] available, open and permanent (with DOI)
Methods
Preprocessing
Method, analysis, processing
Computational environment
same criteria
[level] Methods criteria
[0] unavailable (including available on request)
[1] documented (text, pseudo code, workflow description, versions, Dockerfile, Vagrantfile)
[2] available (source code online, e.g. Github; referring to specific example from paper)
[3] available and open (runtime image/container, standardised metadata, open license)
Results
[level] Results criteria
[0] unavailable/insufficient
[1] documented (understandable, context provided), i.e. reasonable statistical measures/summaries, textual descriptions, tables, maps
[2] available, i.e. models, "output data", scripted plots/maps
[3] available, open and permanent

Figure A.1: Reproducibility criteria to be assessed.

Criteria	Score	Comments
Input data	3	All the CityJSON datasets are collected from open websites.
Preprocessing	2	No specific preprocessing is needed. Once the data is downloaded, it can be inserted into the database by the online source code.
Methods	2	Source code is hosted on GitHub.
Computational environment	3	Only Python and related packages are required.
Results	2	Benchmarks are accssile on GitHub.

Table A.1: The score according to the reproducibility criteria. The source code and benchmarks are available in GitHub.