MSc thesis in Geomatics

3DCITYDB-TOOLS PLUG-IN FOR QGIS:
Adding server-side support to 3DCityDB v.5.0

Bing-Shiuan Tsai

2024

TU Delft

MSc thesis in Geomatics

3DCityDB-Tools plug-in for QGIS: Adding server-side support to 3DCityDB v.5.0

Bing-Shiuan Tsai

October 2024

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:

TU Delft 3Dgeoinfo	3D geoinformation group Delft University of Technology
virtual city systems	Virtual City Systems Berlin, Germany
Hochschule für Technik Stuttgart	Stuttgart University of Applied Sciences Stuttgart, Germany

Delft University of Technology

Supervisors:	Giorgio Agugiaro
	Camilo León Sánchez
Co-reader:	Martijn Meijers

Virtual City System

External Supervisors:	Claus Nagel

Stuttgart University of Applied Sciences

External Supervisors:	Zhihang Yao

Abstract

Semantic 3D city models are essential for visualising, analysing, and managing the built environment. CityGML is an international standard for representing 3D spatial information with a Unified Modelling Language (UML)-based data model to address data heterogeneity and facilitate data exchange. Common formats for encoding CityGML data include Extensible Markup Language (XML), CityJSON, and relational databases, with PostgreSQL preferred for its spatial data management capabilities enhanced by PostGIS.

Although relational databases like 3D City Database (3DCityDB) support CityGML v.1.0 and 2.0, challenges remain due to complex schemas and the need for advanced Structured Query Language (SQL) knowledge to access nested features and attributes of the encoded CityGML data, especially through Geographical Information System (GIS) software like QGIS, which is widely used by Architecture, Engineering and Construction (AEC) professionals. To address these issues, the 3DCityDB-Tools plug-in for QGIS (plug-in) developed by the 3D Geoinformation group at TU Delft [1] simplifies interactions with 3DCityDB-encoded data by providing a user-friendly QGIS interface, enabling the creation of GIS layers composed of unique feature geometries and associated with attributes. With the release of CityGML v.3.0 in 2021, 3DCityDB is being updated to version 5.0, requiring corresponding changes to the plug-in for compatibility.

This thesis investigates the changes in the CityGML spatial concepts and the differences in 3DCityDB encoding. The methods are derived based on the 3DCityDB v.5.0 structure, which consists of schema-wise scans for checking the existing feature geometries and attributes. The scan results are then stored in the metadata tables for users to select the desired feature geometries and attributes for generating GIS layers. In the implementation, feature geometries are determined by the inherited fixed spatial properties of space or boundary features. In contrast, the feature attributes are classified into four types: ”Inline-Single”, ”Inline-Multiple”, ”Nested-Single” and ”Nested-Multiple” according to the modified 3DCityDB encoding. Each type requires specific flattening(linearisation) strategies to be joined with the geometries. Finally, users can generate GIS layers by joining the queried feature geometries and attributes. Several query time performance tests are conducted to determine the method for storing query results and creating the layers.

The generated GIS layers demonstrate flexible access to the feature geometries and attributes with enhanced attribute management. The attribute flattening method facilitates the consumption of complex attributes, making them accessible for batch querying in QGIS. While direct editing of geometries and attributes in GIS layers is not yet supported, these advancements increase the usability of CityGML data. Coping with the XML complex feature schema is a persistent technical challenge in the GIS applications; the proposed approach provides promising alternatives that align with the ongoing development efforts in the QGIS community, offering a complementary pathway for handling complex geospatial data.

Acknowledgements

This nearly year-long graduation project has finally come to an end. First, I would like to express my sincere appreciation to my supervisors, Giorgio Agugiaro and Camilo León Sánchez, for their unwavering support throughout this research. I am truly grateful for their expertise and guidance in exploring CityGML standards and developing strategies to adapt the server side of the QGIS plug-in to the new 3D City Database schema.

I would also like to thank my co-reader, Martijn Meijers, for his valuable feedback and suggestions, which helped me refine this thesis. His insights inspired me to present my results in a clearer, more accessible way. I am also grateful to my external supervisors, Claus Nagel and Zhihang Yao, for providing access to the powerful 3D City Database and their technical insights. Database design is an intricate and fascinating topic, one certainly worth further exploration.

I want to express my heartfelt gratitude to my family in Taiwan for always being my strongest support, especially to my parents for their financial support. Cheers to my Taiwanese and Geomatics friends for their encouragement throughout my studies in the Netherlands. Lastly, to my best friend, I-Tan, it has been a pleasure to meet you here. Thank you for bringing so many surprises, listening to me, and giving me the courage to face life's challenges.

Studying at Delft University of Technology has been on my bucket list since my first visit to the Planning & Design with Water summer school in 2015, and I am proud to make it happen myself. After working as an urban planner in the engineering consultancy realm, I realised the pivotal role of geoinformatics in the planning and design process. Clear, easily accessible spatial data not only facilitates effective communication but also enhances decision-making and data analysis across diverse professional backgrounds. I am glad to contribute to facilitating spatial data usage within the QGIS community, as I have always wanted to since first using QGIS to address real-world issues at work. The solid training and lovely support from the Geomatics folks here in Delft have broadened my horizons and helped me grow immensely. I sincerely thank everyone who has been part of this enriching journey.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

3DCityDB	3D City Database	v
ADE	Application Domain Extension	2
AEC	Architecture, Engineering and Construction	v
COLLADA	COLLAborative Design Activity	1
CSV	Comma-Separated Values	16
EAV model	Entity-Attribute-Value Model	18
ESRI	Environmental System Research Institute	3
GIS	Geographical Information System	v
glTF	Graphics Language Transmission Format	1
GML	Geography Markup Language	8
GMLAS	QGIS GML Application Schema Toolbox plugin	149
GUI	Graphical User Interface	3
IFC	International Foundation Class	10
IoT	Internet of things	3
IMMV	Incrementally Maintainable Materialized View	145
IVM	Incremental View Maintenance	145
JSON	JavaScript Object Notation	1
KML	Keyhole Markup Language	1
LoD	Levels of Detail	xi
LTR	Long Term Release	4
OGC	Open Geospatial Consortium	1
plug-in	3DCityDB-Tools plug-in for QGIS	v
PL/pgSQL	SQL Procedural Language	5
RDBMS	Relational Database Management System	1
SDBMS	Spatially-extended Relational Database Management System	15
SRID	Spatial Reference Identifier	48
SFS model	Simple Feature for SQL Model	3
SQL	Structured Query Language	v
TIN	triangular irregular network	125
ToC	Table of Content	33
UML	Unified Modelling Language	v
UoM	Unit of Measure	64
WFS	Web Feature Service	36
WFS-T	WFS with Transactions	149
WKB	Well-Known Binary	80
XML	Extensible Markup Language	v

1. Introduction

1.1. Motivation

Semantic 3D city models are commonly used for data visualisation and analyses in the built environment. A 3D City Model is a source of harmonised and integrated information for various application domains, including urban planning, energy demand assessment, and environmental management [2]. Storage and management of semantic 3D city model data can be achieved in CityGML, an international standard adopted by the Open Geospatial Consortium (OGC) [3].

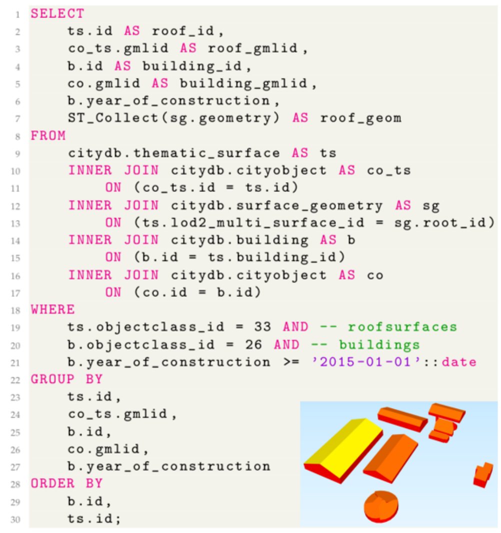
CityGML represents city objects as a standardised UML-based conceptual data model to address the challenges of data heterogeneity, which stems from diverse data sources and storage formats. It caters to various use cases where data interchangeability is required. The principle of CityGML focuses on accommodating and exchanging spatial and non-spatial data related to semantic 3D city models while maintaining hierarchical relations between city objects [3]. For example, buildings can have building parts or other elemental components such as walls and roofs with their specific attributes.

Several data encodings exist for CityGML data. Two of the most common formats are the file-based XML [4] and JavaScript Object Notation (JSON), commonly known as CityJSON, which is a compact subset of the CityGML data model [5]. A third option is the use of Relational Database Management Systems (RDBMSs). Due to the large size of 3D city models, relational databases are the most suitable solution for managing a large amount of spatial data. Relational databases are organised collections of structured data, typically modelled in rows and columns within tables related to one another using a unique ID called the key [6]. An RDBMS is software that provides an interface between users, applications, and databases, enabling users to access and manage the data stored in relational databases [7]. Commonly used RDBMSs include PostgreSQL [8] and Oracle [9]; however, only PostgreSQL is applied in this research due to its powerful open-source solution for spatial data management when combined with the PostGIS [10] extension.

For importing CityGML data, users can utilise ”cjdb”, an open-source solution for storing CityJSON files in PostgreSQL [11], or another open-source RDBMS called 3DCityDB [12, 13]. 3DCityDB is developed for PostgreSQL, Oracle, and PolarDB/Ganos relational databases. Its data schema implements the CityGML standard with detailed semantics and multi-level representations of city objects. In addition to the database schema, the 3DCityDB is equipped with a suite of tools known as the “3DCityDB Suite” [14], which facilitates data import from XML-based CityGML and CityJSON files into the database and data export in Keyhole Markup Language (KML), COLLAborative Design Activity (COLLADA), and Graphics Language Transmission Format (glTF) formats for visualisation in Google Earth and CesiumJS [15]. In contrast, while ”cjdb” focuses on reducing hard disk space consumption by minimising the number of relational tables and extensively using JSON blobs, this approach limits the use of PostGIS functions in PostgreSQL. Consequently, 3DCityDB is used in this research due to its comprehensive support for CityGML data.

The current 3DCityDB in use is the version of 4.x, derived from the CityGML v.2.0 conceptual model. The database imports the CityGML files and converts the city object data and its associations to predefined tables based on the mapping rules consistent with CityGML2.0 standards. The data encoded in this way has several benefits for data application. Firstly, users can directly interact with the data stored in 3DCityDB-standardised tables via SQL queries without the hassle of parsing XML or JSON files [15], and these queries can be improved in terms of search time efficiency from the embedded indices, e.g. B-Tree, supported in PostgreSQL to cope with large dataset. Secondly, users can define functions tailored for specific applications using procedural languages like PL/pgSQL. Lastly, it enhances the usability of CityGML data by allowing users to define different types of Application Domain Extension (ADE) according to specific extension rules [13].

Although the 3DCityDB tool was developed to manage the usage of CityGML data, its structure remains rather complex for users with basic SQL skills. For example, the current 3DCityDB v.4.x consists of 66 predefined tables for storing the features and their relations. The attributes related to the same CityObject class (e.g. Building) are scattered across multiple linked tables with sets of primary and foreign keys to manage their relations. Moreover, CityGML allows for nested features and complex data type attributes, which are nested feature properties that contain other elemental attributes to represent data collections [16]. For instance, buildings can have various geometry LoDs and representations, such as LoD1 solid or LoD2 multi-surface, etc. Additionally, buildings in LoD2 multi-surface can be further represented by aggregating their thematic surfaces, i.e. walls, roofs and grounds, which are structured as nested features. As for the feature attributes, the measured height of the buildings combines two basic types: the double height values and the measuring unit in strings as complex attributes to express the height of the building. The query shown in Figure 1.1 offers an example to retrieve building roofs built since 2015 from the 3DCityDB, showing that all related tables must be joined and geometries must be collected to achieve the desired output [15].



```sql
SELECT
    ts.id AS roof_id,
    co_ts.gmlid AS roof_gmlid,
    b.id AS building_id,
    co.gmlid AS building_gmlid,
    b.year_of_construction,
    ST_Collect(sg.geometry) AS roof_geom
FROM
    citydb.thematic_surface AS ts
    INNER JOIN citydb.cityobject AS co_ts
        ON (co_ts.id = ts.id)
    INNER JOIN citydb.surface_geometry AS sg
        ON (ts.lod2_multi_surface_id = sg.root_id)
    INNER JOIN citydb.building AS b
        ON (b.id = ts.building_id)
    INNER JOIN citydb.cityobject AS co
        ON (co.id = b.id)
WHERE
    ts.objectclass_id = 33 AND -- roofsurfaces
    b.objectclass_id = 26 AND -- buildings
    b.year_of_construction >= '2015-01-01'::date
GROUP BY
    ts.id,
    co_ts.gmlid,
    b.id,
    co.gmlid,
    b.year_of_construction
ORDER BY
    b.id,
    ts.id;
```

Figure 1.1.: Example of the query to extract the roofs of the buildings built since 2015 from the data stored within 3DCityDB (figure from [15])

Since most users work with spatial data, the AEC practitioners nowadays are most likely using the application GIS. The complex features of CityGML data stored in 3DCityDB do not follow the Simple Feature for SQL Model (SFS model)[17], which is a vector representation adopted by most of the GIS software, for instance, the ArcGIS from the Environmental System Research Institute (ESRI) and open-source QGIS. The SFS model structures a feature class as a table that takes different features as an entry with unique identifiers followed by columns of its geometry and attributes [18]. The factors mentioned above limit the access of CityGML data stored in 3DCityDB, implying that a user would have to be equipped with advanced knowledge of both SQL and the 3DCityDB structure for processing the data to consume the nested features and complex data type attributes following the CityGML model through GIS software [15].

To tackle the limitations of 3DCityDB, a plugin called "plug-in" [1] has been developed by the 3D Geoinformation group at Technische Universiteit (TU) Delft to alleviate the complexity of 3DCityDB schemas in the background. It provides a user-friendly, Graphical User Interface (GUI)-based interface directly working within QGIS, which comprehensively takes user inputs, runs corresponding queries on the server-side to perform operations like spatial and semantic filters and structures the result as classic GIS layers. The proposed layers-based concept enables users to interact with the data encoded in 3DCityDB more intuitively, allowing data editing by QGIS tools or plug-in functions [15].

The main purpose of the plug-in is to encourage the use of CityGML/CityJSON data for users with limited experience with SQL and Spatial-RDBMSs, narrowing the gap between them and complex 3D city models, as the management of semantic 3D city models can generally be optimised by spatial databases like 3DCityDB. The user-friendly interface supported by the plug-in is expected to facilitate further development of open-source 3DCityDB by including practitioners from heterogeneous backgrounds with their aids, which thus expands the usability of 3D city model data [15, 19].

In the meanwhile, the OGC published the CityGML v.3.0 standard in September 2021[20], which introduces improved conceptual data models, a revised space and LoD concepts such as logical space and physical space to address the spatial characteristics and the support of time-dependent Internet of things (IoT) data, etc [21]. The 3DCityDB is being updated to version 5.0 to add full support for CityGML v.3.0. However, the current 3DCityDB-Tools plug-in for QGIS only supports CityGML v.1.0 and 2.0. Therefore, the goal of this research is to investigate how the new database structure of the 3DCityDB v.5.0 can be coupled with the existing 3DCityDB-Tools plug-in for QGIS to enable support not only for the existing version 4.x but also for the upcoming version 5.0.

1.2. Research Questions and Objectives

The main objective of this thesis is to explore methods for enhancing the server-side of 3DCityDB-Tools plug-in for QGIS, adapting functionalities to support 3DCityDB v.5.0 and, consequently, CityGML v.3.0. To achieve this goal, this thesis aims to answer the main research question, together with the following specified sub-questions:

How does the new database structure of 3DCityDB v.5.0 affect the current methods of the plugin to create layers which contain both geometries and attributes for a selected feature type following the SFS model? In particular,

1. **How do the new CityGML 3.0 concepts of space, and LoD affect the process?**
2. **Regarding geometries, can the same or a similar approach be reproduced?**
    - **Is it still necessary to rely on the materialized views?**
    - **What alternatives are there?**
3. **Regarding attributes, can the same or a similar approach be reproduced?**
    - **Is it still necessary to rely on updatable views?**
    - **What alternatives are there?**
4. **How is the CityGML v.2.0 data mapped to the new schema of 3DCityDB v.5.0?**
    - **Can we deal with CityGML v.2.0 data as CityGML v.3.0 data as long as it is stored in the 3DCityDB v.5.0?**

1.3. Research Scope

According to the specified research questions, this thesis aims to reproduce the layers-based method implemented by the current plug-in with the new 3DCityDB v.5.0 database structure. The layers generated by the current plug-in for GIS software consist of two parts: the retrieval of feature geometries as the entries followed by their corresponding feature attributes. Therefore, the main focus is on the server-side of the plug-in, providing methods to collect feature geometries and attributes separately and eventually create GIS layers in QGIS for the usage of CityGML data encoded with 3DCityDB v.5.0. The client-side aspect is not entirely outside the scope of this research; suggestions for the client-side, such as GUI modifications, are discussed, as certain adjustments are expected to adapt the plug-in to changes on the server side.

The CityGML v.3.0 data model has 15 thematic modules. The research only focuses on gathering feature geometries and attributes of the classes in the following modules: ”Building”, ”Bridge”, ”Tunnel”, ”Vegetation”, ”CityFurniture”, ”Generics”, ”Relief”, ”LandUse”, ”Transportation” and ”WaterBody”, which are ten modules in total.

The ”Construction”, ”Dynamizer”, ”Versioning” and ”PointCloud” are four newly added modules in the new CityGML v.3.0 standards. At the time of this research, the 3DCityDB v.5.0 does not support the ”PointCloud” module in the database, and its development is still in progress. Considering that the four new modules and the ”CityObjectGroup” and ”Appearance” modules are about additional semantics or modules that are not directly related to urban object types, these six modules are excluded from this research.

The development of this thesis is based on CityGML data encoded by 3DCityDB-Command-line-tool-0.8.1-beta [22] for PostgreSQL/PostGIS, operated with the aid of an open-source management tool for PostgreSQL, pgAdmin 4 v.8.12 and QGIS Long Term Release (LTR) v.3.34.8-Prizren for GIS layers visualisation and operation.

1.4. Thesis Structure

The research begins by familiarising with the mapping structure of the CityGML data to the new data schema in 3DCityDB v.5.0. Methods for collecting and processing feature geometries and attributes are then developed according to the new 3DCityDB encoding. Experiments on geometry and attribute views, or materialized views, are conducted to test query time efficiency for decision-making. Finally, with the generated feature geometry and attribute views, approaches for GIS layer creation are introduced, followed by the discussion on the results and future adaptations of the plug-in tailored for the 3DCityDB v.5.0.

Following this Introduction section, the rest of the thesis is structured as follows:

- **Chapter 2** provides the theoretical and technical background of this thesis, including a brief introduction to CityGML v.2.0, v.3.0 and the fundamental change of the geometry and LoD concept, followed by the 3DCityDB encoding the structural difference between 3DCityDB v.4.x and 5.0. Furthermore, the structure of the current plug-in is covered to introduce its structure and the methods implemented to create SFS model layers for 3D city model data usage in QGIS.
- **Chapter 3** identifies the potential use cases of the CityGML data stored in 3DCityDB v.4.x through QGIS, which clarifies the research objectives for adding plug-in server-side support to the 3DCityDB v.5.0.
- **Chapter 4** elaborates the methodology for checking existing feature geometries and attributes in the target dataset and for collecting feature geometries based on available geometry representations and LoDs. The four attribute classes—”Inline-Single”, ”Inline-Multiple”, ”Nested-Single”, and ”Nested-Multiple”—are introduced to facilitate the collection and flattening (linearisation) of feature attributes encoded within 3DCityDB v.5.0. Subsequently, the approaches for GIS layer creation are presented, detailing the methods used for joining feature geometries with their corresponding attributes.
- **Chapter 5** first provides setup instructions for 3DCityDB v.5.0 and CityGML data import, then explains the implementation process for checking the existence of feature geometries and attributes, treating them as metadata to be collected and stored in PostgreSQL views or materialized views. The implementation details for creating feature geometry and attribute views are presented in SQL templates, demonstrating the dynamic query generation via the SQL Procedural Language (PL/pgSQL) function developed in this research.
- **Chapter 6** proposes three approaches for joining feature geometries and attributes to create GIS layers that comply with the SFS model. This is followed by a layer query performance test to determine the most effective approach. The results of the generated layers using the test datasets are then presented, utilising the chosen approach.
- **Chapter 7** evaluates the achievable use cases based on the methodology and implementation presented in this research and identifies areas for future development of the plug-in.
- **Chapter 8** reveals the conclusions, discussion, and limitations, followed by the proposal for future development.

2. Related work

This chapter provides an overall review of the related background relevant to this research. Firstly, the CityGML data model is introduced (Section 2.1). Secondly, the 3DCityDB structure is explained with a focus on the feature geometries and attributes of their encoding in the data schema, which are the pre-defined tables that are queried as the source tables (Section 2.2). Thirdly, the functionalities of "view" and "materialized view" methods supported by RDBMS are covered as these functions form the technical base for GIS layer creation provided by the current plug-in (Section 2.3). Finally, the review of the current plug-in is covered with the server-side and client-side structures and the tests and limitations, giving the introduction of the proposed layer methodology (Section 2.4).

2.1. CityGML

The current plug-in built for the usage of CityGML data encoded in 3DCityDB supports both CityGML v.1.0 and v.2.0. However, the data model version commonly in use nowadays is the CityGML v.2.0. Thus only this standard version is introduced. Moreover, the CityGML v3.0 standards released by the OGC in 2021 introduced significant changes, particularly in feature geometry and LoD concepts. The following paragraphs give an overview of the characteristics of CityGML v.3.0 together with the main difference from CityGML v.2.0 in terms of geometry and LoD concepts.

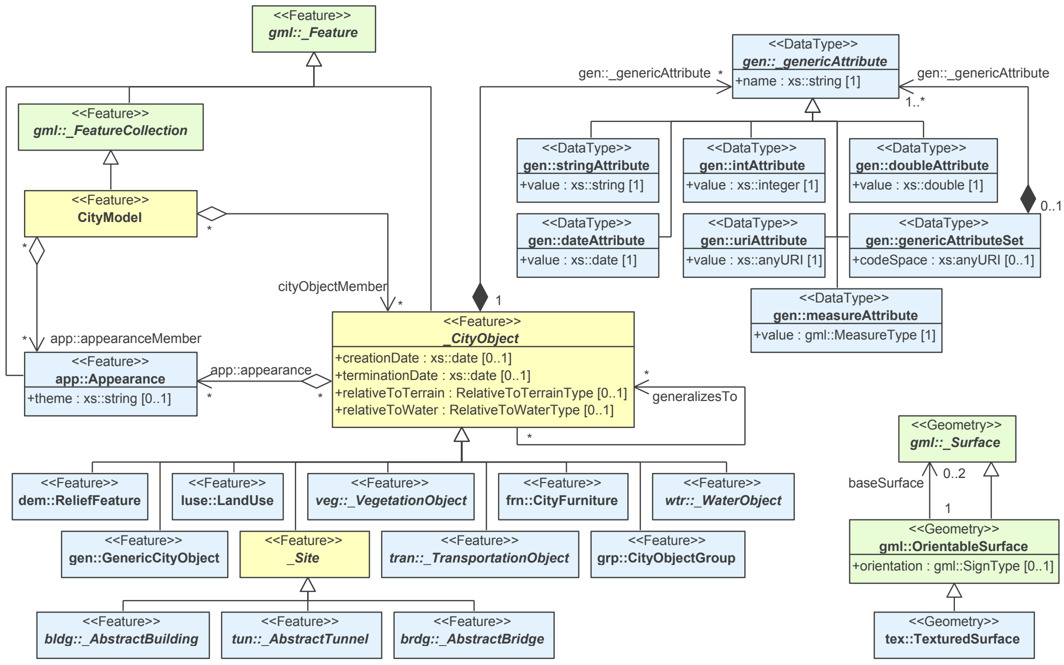
CityGML defines the classes and relations for the most relevant topographic objects in cities and regional models concerning their geometrical, topological, semantic, and appearance properties. It allows visualisation for 3D models and enables thematic queries, analysis tasks, or spatial data mining, satisfying the information needs of various application fields [20]. The principle of the CityGML is to represent reality with all the different types of urban objects and their relations, which is achieved by grouping different object classes in specific thematic modules (Figure 2.3). For example, the "Building" module contains all relevant sub-classes used to model buildings, etc. All thematic modules import the Core module with common properties, representing how object characteristics and their relations are set in reality.

2.1.1. CityGML v.2.0

CityGML v.2.0 was published in 2012 as an update of the CityGML v.1.0 standard, first released in 2008. The structure of CityGML v.2.0 follows a core-thematic modular hierarchy, where all classes in the thematic modules are derived from the Core module, sharing common attributes.

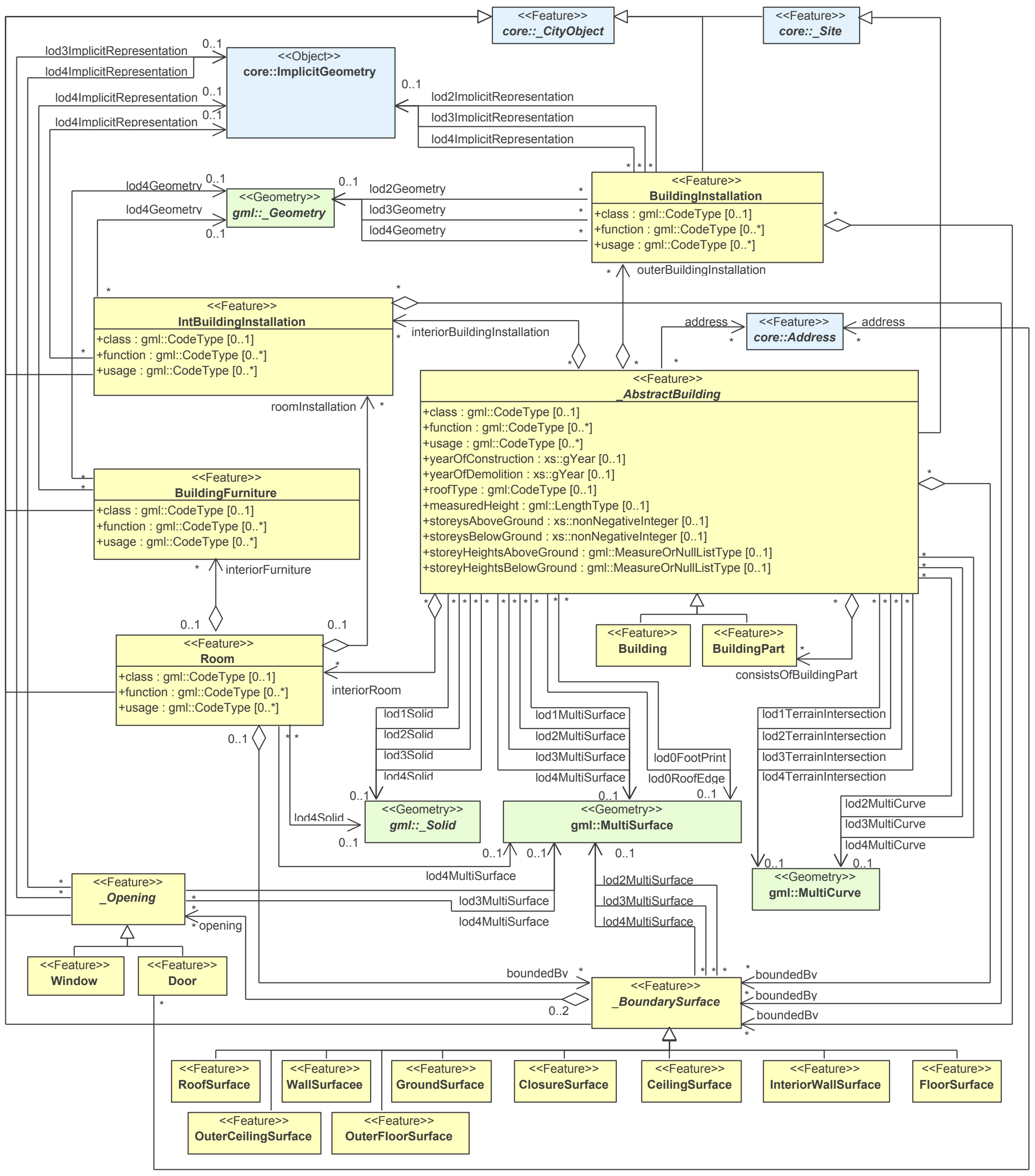
In CityGML v.2.0, the Core module defines the basic feature class, which is the abstract class "_CityObject" as all classes in the thematic modules are derived from this starting point abstract class (Figure 2.1). The spatial properties of city objects are represented by objects of Geography Markup Language (GML)3's geometry model based on the standard ISO 19107 "Spatial schema" [23], representing 3D geometry according to the well-known Boundary Representation (B-Rep, cf.[24])[3]. The feature geometry and LoD concepts are associated with the thematic modules, for example, the Building module in Figure 2.2, indicating that the geometry representation of a feature is determined by the degree of its semantic decomposition [25]. For instance, the geometry of a building can be represented by a prismatic geometry (Solid) in LoD1 or by an aggregated geometry (MultiSuface) of its thematic component geometries such as wall, roof and ground surfaces in LoD2 or have more details in LoD3 [15]. The primitive geometry type in CityGML v.2.0 is planar polygons [26], implying that the corresponding component polygons need to be collected when referencing each feature geometry representation.

Moreover, feature geometry representations vary in LoDs according to the specification of different thematic object classes, limiting the available geometry representations of certain features. For instance, the Building thematic surface features are only available starting from LoD2, and the interior Building Room features are only available in LoD4 [25].



<<Feature>> gml::_Feature
<<Feature>> gml::_FeatureCollection
<<Feature>> CityModel
cityObjectMember
app::appearanceMember
<<Feature>> app::Appearance
+theme : xs::string [0..1]
app::appearance
<<Feature>> _CityObject
+creationDate : xs::date [0..1]
+terminationDate : xs::date [0..1]
+relativeToTerrain : RelativeToTerrainType [0..1]
+relativeToWater : RelativeToWaterType [0..1]
generalizesTo
gen::_genericAttribute
<<DataType>> gen::_genericAttribute
+name : xs::string [1]
1..*
0..1
<<DataType>> gen::stringAttribute
+value : xs::string [1]
<<DataType>> gen::intAttribute
+value : xs::integer [1]
<<DataType>> gen::doubleAttribute
+value : xs::double [1]
<<DataType>> gen::dateAttribute
+value : xs::date [1]
<<DataType>> gen::uriAttribute
+value : xs::anyURI [1]
<<DataType>> gen::genericAttributeSet
+codeSpace : xs:anyURI [0..1]
<<DataType>> gen::measureAttribute
+value : gml::MeasureType [1]
<<Feature>> dem::ReliefFeature
<<Feature>> luse::LandUse
<<Feature>> veg::_VegetationObject
<<Feature>> frn::CityFurniture
<<Feature>> wtr::_WaterObject
<<Feature>> gen::GenericCityObject
<<Feature>> _Site
<<Feature>> tran::_TransportationObject
<<Feature>> grp::CityObjectGroup
<<Feature>> bldg::_AbstractBuilding
<<Feature>> tun::_AbstractTunnel
<<Feature>> brdg::_AbstractBridge
<<Geometry>> gml::_Surface
baseSurface
0..2
1
<<Geometry>> gml::OrientableSurface
+orientation : gml::SignType [0..1]
<<Geometry>> tex::TexturedSurface

Figure 2.1.: CityGML v.2.0 Package UML diagram - Core module, part 2 (Figure from [3])



<<Feature>> core::_CityObject
<<Feature>> core::_Site
<<Object>> core::ImplicitGeometry
<<Geometry>> gml::_Geometry
<<Feature>> BuildingInstallation
+class : gml::CodeType [0..1]
+function : gml::CodeType [0..*]
+usage : gml::CodeType [0..*]
<<Feature>> IntBuildingInstallation
<<Feature>> core::Address
<<Feature>> _AbstractBuilding
+class : gml::CodeType [0..1]
+function : gml::CodeType [0..*]
+usage : gml::CodeType [0..*]
+yearOfConstruction : xs::gYear [0..1]
+yearOfDemolition : xs::gYear [0..1]
+roofType : gml::CodeType [0..1]
+measuredHeight : gml::LengthType [0..1]
+storeysAboveGround : xs::nonNegativeInteger [0..1]
+storeysBelowGround : xs::nonNegativeInteger [0..1]
+storeyHeightsAboveGround : gml::MeasureOrNullListType [0..1]
+storeyHeightsBelowGround : gml::MeasureOrNullListType [0..1]
<<Feature>> BuildingFurniture
<<Feature>> Room
<<Feature>> Building
<<Feature>> BuildingPart
consistsOfBuildingPart
<<Geometry>> gml::_Solid
<<Geometry>> gml::MultiSurface
<<Geometry>> gml::MultiCurve
<<Feature>> _Opening
<<Feature>> Window
<<Feature>> Door
<<Feature>> _BoundarySurface
<<Feature>> RoofSurface
<<Feature>> WallSurfacee
<<Feature>> GroundSurface
<<Feature>> ClosureSurface
<<Feature>> CeilingSurface
<<Feature>> InteriorWallSurface
<<Feature>> FloorSurface
<<Feature>> OuterCeilingSurface
<<Feature>> OuterFloorSurface
lod2ImplicitRepresentation
lod3ImplicitRepresentation
lod4ImplicitRepresentation
lod2Geometry
lod3Geometry
lod4Geometry
outerBuildingInstallation
interiorBuildingInstallation
roomInstallation
interiorFurniture
interiorRoom
address
lod1Solid
lod2Solid
lod3Solid
lod4Solid
lod1MultiSurface
lod2MultiSurface
lod3MultiSurface
lod4MultiSurface
lod0FootPrint
lod0RoofEdge
lod1TerrainIntersection
lod2TerrainIntersection
lod3TerrainIntersection
lod4TerrainIntersection
lod2MultiCurve
lod3MultiCurve
lod4MultiCurve
opening
boundedBy

Figure 2.2.: CityGML v.2.0 Package UML diagram - Building module (Figure from [3])

2.1.2. CityGML v.3.0

CityGML v.3.0 was introduced in 2021 by the OGC as a direct evolution of CityGML v.2.0. It follows a similar structure to CityGML v.2.0, where the thematic modules inherit common properties from the Core module (Figure 2.3). Notable changes include the introduction of the ”Construction” module, which is imported by the ”Building”, ”Bridge”, and ”Tunnel” modules for better interoperability with other standards such as International Foundation Class (IFC) and the addition of three new modules: ”Dynamizer”, ”PointCloud”, and ”Versioning”. These newly added modules are designed to address IoT data, spatial data represented in point clouds, and changes in a city model, respectively.

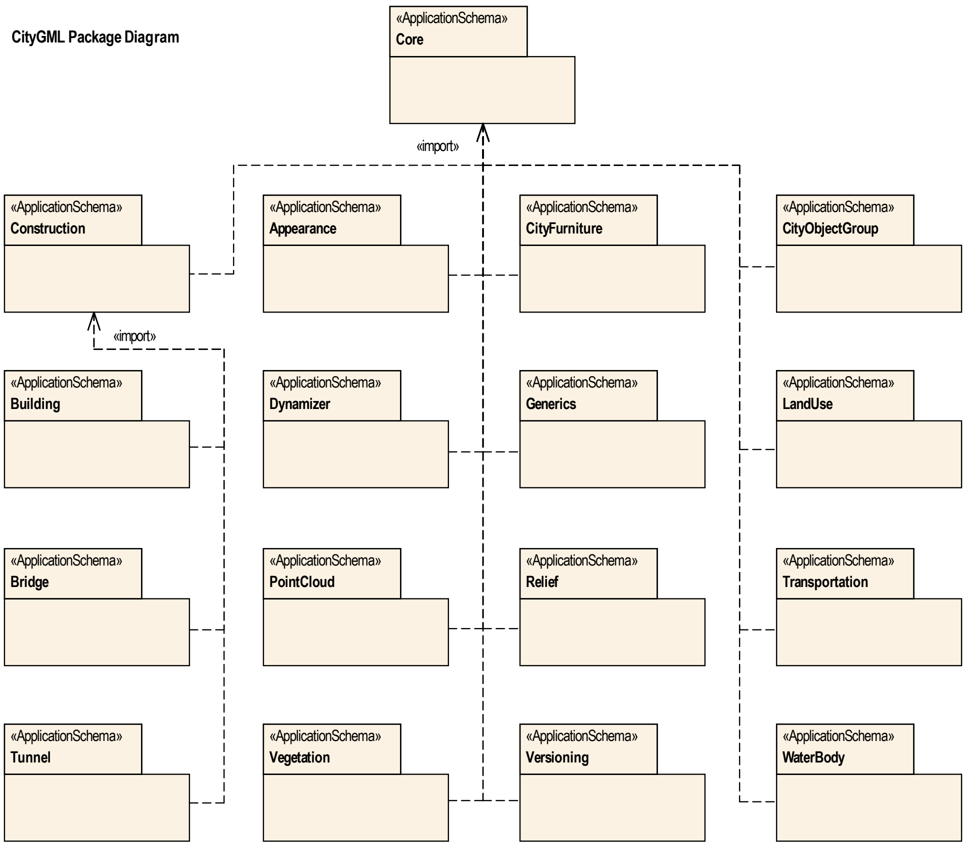

CityGML Package Diagram
«ApplicationSchema» Core
«import»
«ApplicationSchema» Construction
«ApplicationSchema» Appearance
«ApplicationSchema» CityFurniture
«ApplicationSchema» CityObjectGroup
«import»
«ApplicationSchema» Building
«ApplicationSchema» Dynamizer
«ApplicationSchema» Generics
«ApplicationSchema» LandUse
«ApplicationSchema» Bridge
«ApplicationSchema» PointCloud
«ApplicationSchema» Relief
«ApplicationSchema» Transportation
«ApplicationSchema» Tunnel
«ApplicationSchema» Vegetation
«ApplicationSchema» Versioning
«ApplicationSchema» WaterBody

Figure 2.3.: CityGML v.3.0 Package UML diagram showing the dependencies of the Core modules and all the thematic modules (Figure from [20])

The Core module composition of CityGML v.3.0 includes:

- City models represent real-world cities and landscapes, which can be represented by aggregating different types of city objects, appearances, feature objects, etc. All objects defined in the CityGML v.3.0 standard are featured with lifespans, allowing optional existence specifications in real-world and database times. City objects are the features that define thematic concepts in more detail properties like buildings, bridges, land use, etc. [20].

- Space concepts are a clear semantic distinction of spatial features, which are newly introduced in CityGML v.3.0. It differentiates all city objects into spaces and space boundaries. According to [21], “Spaces are entities of volumetric extent in the real world”; for example, buildings, trees and traffic space have volumetric extent. “Space boundaries are entities with the areal extent in the real world”, which can be differentiated into different types of thematic surfaces, such as wall surfaces and roof surfaces. Spaces can be further classified into physical spaces and logical spaces; physical spaces are used to represent the volumetric extent bounded by physical objects in space, while logical spaces are defined according to the thematic consideration, such as a building unit aggregated by specific rooms to flats as a whole. The space concepts are crucial characteristics in the CityGML v.3.0 Core module as the feature geometries are associated with the “AbstractSpace” and “AbstractSpaceBoundary” classes and the deriving “AbstractThematicSurface” class shown in Figure 2.4.

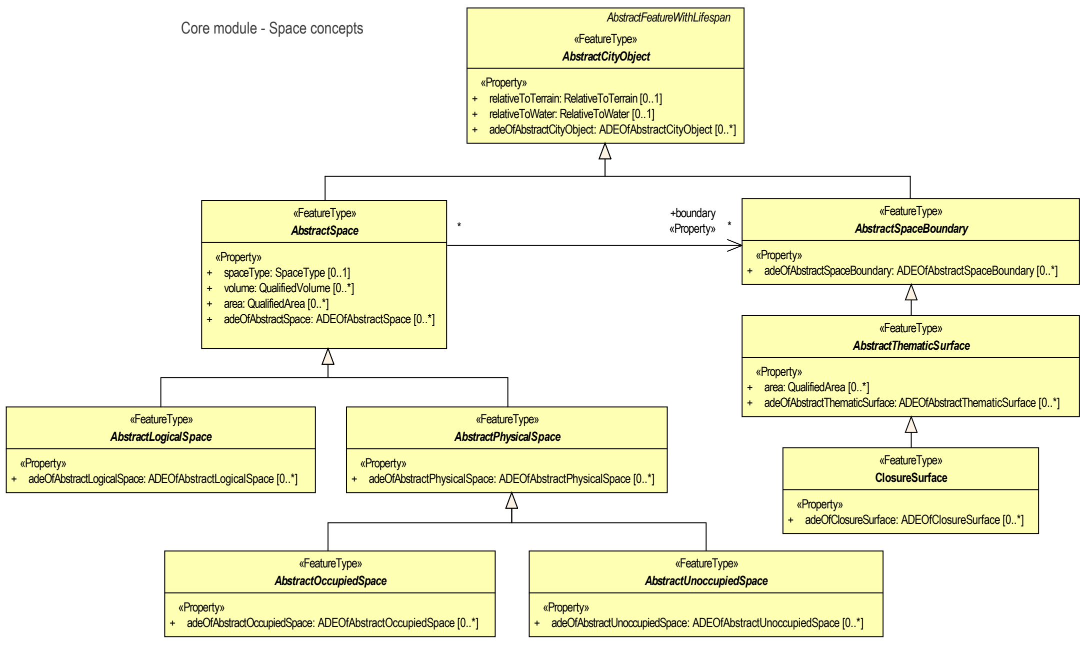

Core module - Space concepts
AbstractFeatureWithLifespan
«FeatureType»
AbstractCityObject
«Property»
+ relativeToTerrain: RelativeToTerrain [0..1]
+ relativeToWater: RelativeToWater [0..1]
+ adeOfAbstractCityObject: ADEOfAbstractCityObject [0..*]
«FeatureType»
AbstractSpace
«Property»
+ spaceType: SpaceType [0..1]
+ volume: QualifiedVolume [0..*]
+ area: QualifiedArea [0..*]
+ adeOfAbstractSpace: ADEOfAbstractSpace [0..*]
*
+boundary
«Property»
*
«FeatureType»
AbstractSpaceBoundary
«Property»
+ adeOfAbstractSpaceBoundary: ADEOfAbstractSpaceBoundary [0..*]
«FeatureType»
AbstractThematicSurface
«Property»
+ area: QualifiedArea [0..*]
+ adeOfAbstractThematicSurface: ADEOfAbstractThematicSurface [0..*]
«FeatureType»
ClosureSurface
«Property»
+ adeOfClosureSurface: ADEOfClosureSurface [0..*]
«FeatureType»
AbstractLogicalSpace
«Property»
+ adeOfAbstractLogicalSpace: ADEOfAbstractLogicalSpace [0..*]
«FeatureType»
AbstractPhysicalSpace
«Property»
+ adeOfAbstractPhysicalSpace: ADEOfAbstractPhysicalSpace [0..*]
«FeatureType»
AbstractOccupiedSpace
«Property»
+ adeOfAbstractOccupiedSpace: ADEOfAbstractOccupiedSpace [0..*]
«FeatureType»
AbstractUnoccupiedSpace
«Property»
+ adeOfAbstractUnoccupiedSpace: ADEOfAbstractUnoccupiedSpace [0..*]

Figure 2.4.: CityGML v.3.0 Package UML diagram - Core module, Space concepts (Figure from [20])

- Geometry and LoD concepts define the spatial properties, including the geometry data types and the geometry LoDs of the CityGML features. The geometry and LoD concepts are explained in more detail in Section 2.1.1 and Section 2.1.2 as they are highly relevant to the feature geometry mapping in the 3DCityDB and the feature geometry extraction of the current plug-in.

- Enumerations are the fixed list of named literal values defined by CityGML standards, which consists of standardised valid values used for feature attributes.

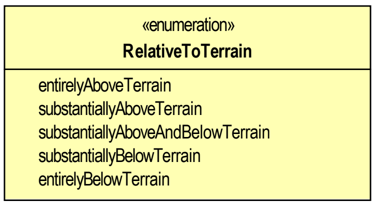

«enumeration»
RelativeToTerrain
entirelyAboveTerrain
substantiallyAboveTerrain
substantiallyAboveAndBelowTerrain
substantiallyBelowTerrain
entirelyBelowTerrain

Figure 2.5.: CityGML v.3.0 Package UML diagram Core module - an enumeration example of "CityObject"'s "RelativeToTerrain" attribute valid values (Figure from [20])

- Codelists are the list of customised-defined values not in line with CityGML standards. For instance, the "BuildingClassValue" from the OGC CityGML v.3.0 codelist repository [27]

The CityGML v.3.0 Core module defines Space concepts and the basic abstract class, "AbstractCityObject", the starting point abstract class of all classes in the thematic modules (Figure 2.4). The Space concepts are new semantic classes derived from "AbstractCityObject", which consists of two abstract classes, "AbstractSpace" and " AbstractSpaceBoundary", for volumetric and areal extent, respectively. There are four deriving classes from "AbstractSpace", starting from the first derivation of "AbstractPhysicalSpace" and "AbstractLogicalSpace" to classify physical and logical spaces to the second derivation of "AbstractOccupiedSpace" and " AbstractUnoccupiedSpace" to categorise physical space into physically occupied and unoccupied spaces (Figure 2.4). The concrete classes in thematic modules such as Building, Vegetation or Transportation inherit the common attributes from these abstract classes. From the "AbstractSpaceBoundary" class, the "AbstractThematicSurface" class is derived, which is the abstract class for all concrete surface classes like "ClosureSurface" or thematic surface classes in the Construction module like "WallSurface", "RoofSurface", etc. [21]

The spatial properties of CityGML v.3.0 features also utilise geometry data types from ISO 19107 [28]. However, the geometry and LoD concepts are specifically associated with the classes "AbstractSpace" and "AbstractThematicSurface" in the Core module, as shown in Figure 2.6. These two abstract classes possess geometry representations in different LoDs, which are inherited by the thematic modules, and the geometries are directly stored without decomposition. The modified spatial and LoD concept eliminates LoD4 and allows features to be represented in mixed LoDs. For example, buildings can have their outer shell represented in LoD2 and inner rooms represented in LoD0 floorplans. The mixed-use of different LoDs supports using 3D city models in applications that require detailed interior representations but not necessarily the exterior, such as indoor navigation and energy analysis. This new spatial and LoD concept enhances the flexibility in representing both the interior and exterior of features and simplifies the geometry representations of all classes in the thematic modules, as they share identical geometry and LoD concepts with the space and space boundary abstract classes, except for the Relief module.

Since the Relief module is derived from the "AbstractSpaceBoundary" class, which is the higher level abstract class of the "AbstractThematicSurface" class, the features in the Relief module have their specified geometry and LoD concepts (Figure 2.7).

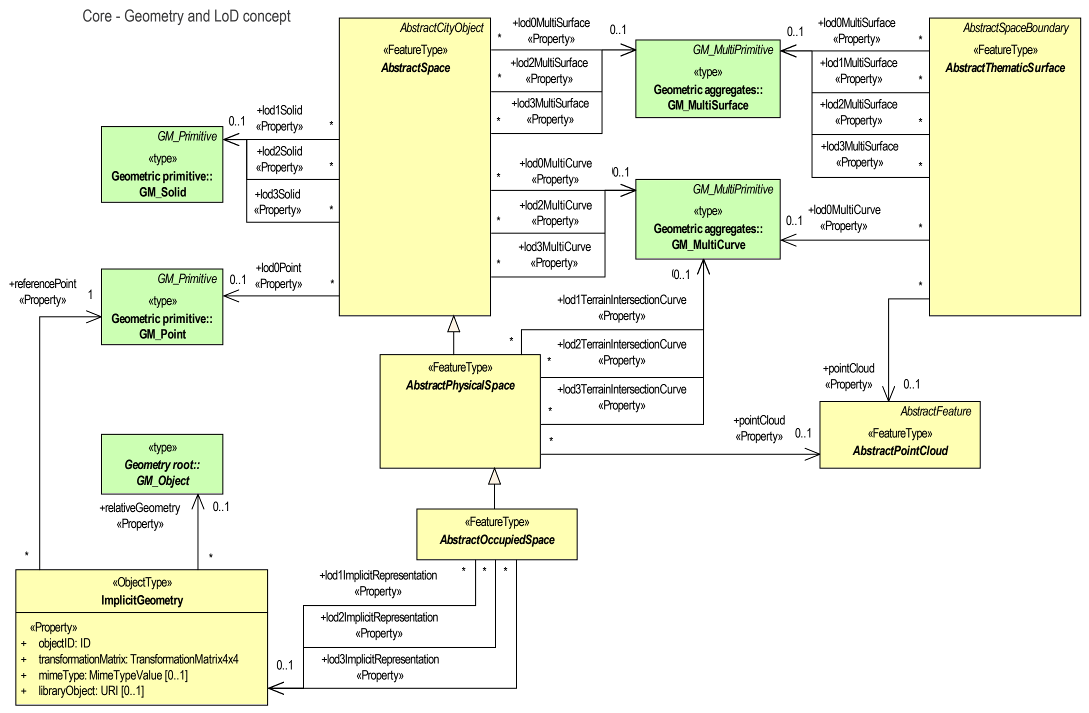

Core - Geometry and LoD concept
AbstractCityObject
«FeatureType»
AbstractSpace
+lod0MultiSurface «Property»
+lod2MultiSurface «Property»
+lod3MultiSurface «Property»
GM_MultiPrimitive
«type»
Geometric aggregates:: GM_MultiSurface
AbstractSpaceBoundary
«FeatureType»
AbstractThematicSurface
+lod0MultiSurface «Property»
+lod1MultiSurface «Property»
+lod2MultiSurface «Property»
+lod3MultiSurface «Property»
+lod1Solid «Property»
+lod2Solid «Property»
+lod3Solid «Property»
GM_Primitive
«type»
Geometric primitive:: GM_Solid
+lod0MultiCurve «Property»
+lod2MultiCurve «Property»
+lod3MultiCurve «Property»
GM_MultiPrimitive
«type»
Geometric aggregates:: GM_MultiCurve
+lod0MultiCurve «Property»
+referencePoint «Property»
+lod0Point «Property»
GM_Primitive
«type»
Geometric primitive:: GM_Point
+lod1TerrainIntersectionCurve «Property»
+lod2TerrainIntersectionCurve «Property»
+lod3TerrainIntersectionCurve «Property»
«FeatureType»
AbstractPhysicalSpace
+pointCloud «Property»
+pointCloud «Property»
AbstractFeature
«FeatureType»
AbstractPointCloud
«type»
Geometry root:: GM_Object
+relativeGeometry «Property»
«FeatureType»
AbstractOccupiedSpace
«ObjectType»
ImplicitGeometry
«Property»
+ objectID: ID
+ transformationMatrix: TransformationMatrix4x4
+ mimeType: MimeTypeValue [0..1]
+ libraryObject: URI [0..1]
+lod1ImplicitRepresentation «Property»
+lod2ImplicitRepresentation «Property»
+lod3ImplicitRepresentation «Property»
0..1
1
*

Figure 2.6.: CityGML v.3.0 Package UML diagram - Core module, Geometry and LoD concept (Figure from [20])

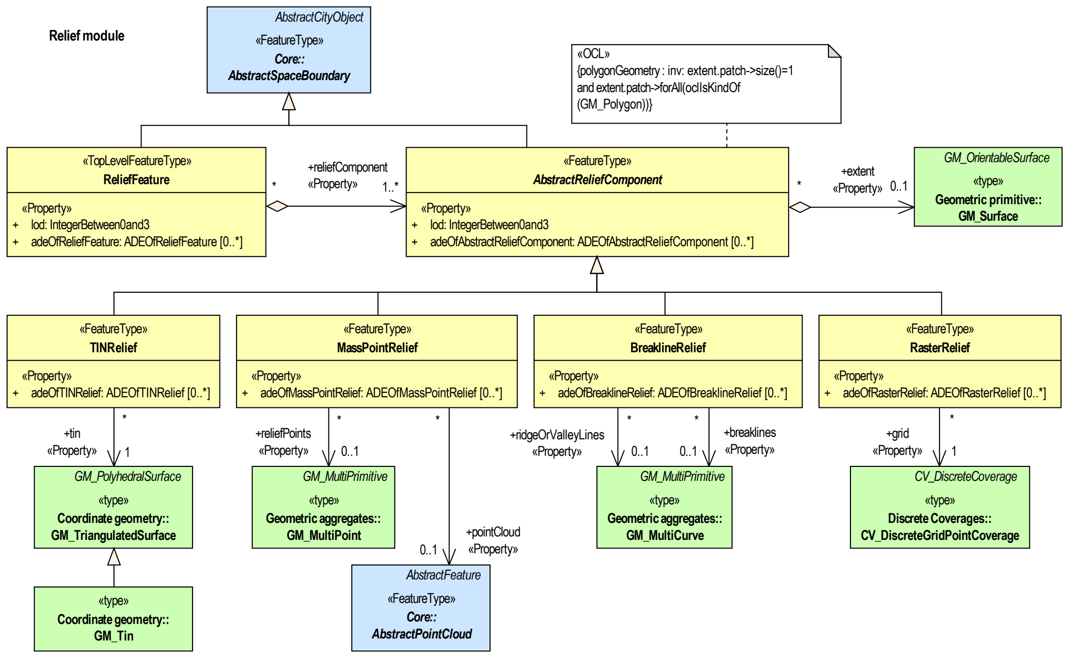

Relief module
AbstractCityObject
«FeatureType»
Core:: AbstractSpaceBoundary
«OCL»
{polygonGeometry : inv: extent.patch->size()=1 and extent.patch->forAll(oclIsKindOf (GM_Polygon))}
«TopLevelFeatureType»
ReliefFeature
«Property»
+ lod: IntegerBetween0and3
+ adeOfReliefFeature: ADEOfReliefFeature [0..*]
+reliefComponent «Property»
1..*
«FeatureType»
AbstractReliefComponent
«Property»
+ lod: IntegerBetween0and3
+ adeOfAbstractReliefComponent: ADEOfAbstractReliefComponent [0..*]
+extent «Property»
0..1
GM_OrientableSurface
«type»
Geometric primitive:: GM_Surface
«FeatureType»
TINRelief
«Property»
+ adeOfTINRelief: ADEOfTINRelief [0..*]
«FeatureType»
MassPointRelief
«Property»
+ adeOfMassPointRelief: ADEOfMassPointRelief [0..*]
«FeatureType»
BreaklineRelief
«Property»
+ adeOfBreaklineRelief: ADEOfBreaklineRelief [0..*]
«FeatureType»
RasterRelief
«Property»
+ adeOfRasterRelief: ADEOfRasterRelief [0..*]
+tin «Property»
+reliefPoints «Property»
+ridgeOrValleyLines «Property»
+breaklines «Property»
+grid «Property»
+pointCloud «Property»
GM_PolyhedralSurface
«type»
Coordinate geometry:: GM_TriangulatedSurface
GM_MultiPrimitive
«type»
Geometric aggregates:: GM_MultiPoint
GM_MultiPrimitive
«type»
Geometric aggregates:: GM_MultiCurve
CV_DiscreteCoverage
«type»
Discrete Coverages:: CV_DiscreteGridPointCoverage
«type»
Coordinate geometry:: GM_Tin
AbstractFeature
«FeatureType»
Core:: AbstractPointCloud

Figure 2.7.: CityGML v.3.0 Package UML diagram - Relief module (Figure from [20])

The geometry representation of the “AbstractSpace” class includes LoD0 ”Point”, LoD1-3 “Solid”, LoD0,2-3 “MultiSurface”, LoD0,2-3 “MultiCurve”, LoD1-3 “TerrainIntersectionCurve”, LoD1-3 “ImplicitRepresentation” and “pointCloud” these 17 geometry types. In the “AbstractThematicSurface” class, the geometry representations include LoD0 “MultiCurve”, LoD0-3 “MultiSurface” and “pointCloud” these six geometry types. The listed geometry types follow the ISO 19170 standards [28]. The “Implicit Representation” is an additional geometry concept that adheres to the ISO 19170 standards. This concept allows for storing prototypical geometry, which can then be re-used or referenced whenever the corresponding feature occurs [20].

The association of geometry and LoD concepts to the Core modules reduces the complexity of features’ geometry representations. In CityGML v.3.0, all features are derived from the core abstract space and space boundary classes, with limited LoDs and geometry representation combinations. For example, the buildings can only be represented in the seventeen geometry types, as the parent classes of the “Building” class are derived from the “AbstractSpace” class (Figure 2.8). The revised geometry and LoD concepts also make the LoD representations of the thematic surface classes more flexible, allowing classes like the “WallSurface” and “GroundSurface” to have geometry in LoD0-1 while window and doors can be represented in all LoD0-3 [25] as they are all inherit the six geometry types from “AbstractThematicSurface” class.

For convenience, throughout this research, features derived from the ”AbstractSpace” class will be referred to as ”space features.” In contrast, features derived from the ”AbstractThematicSurface” class will be referred to as ”boundary features.”

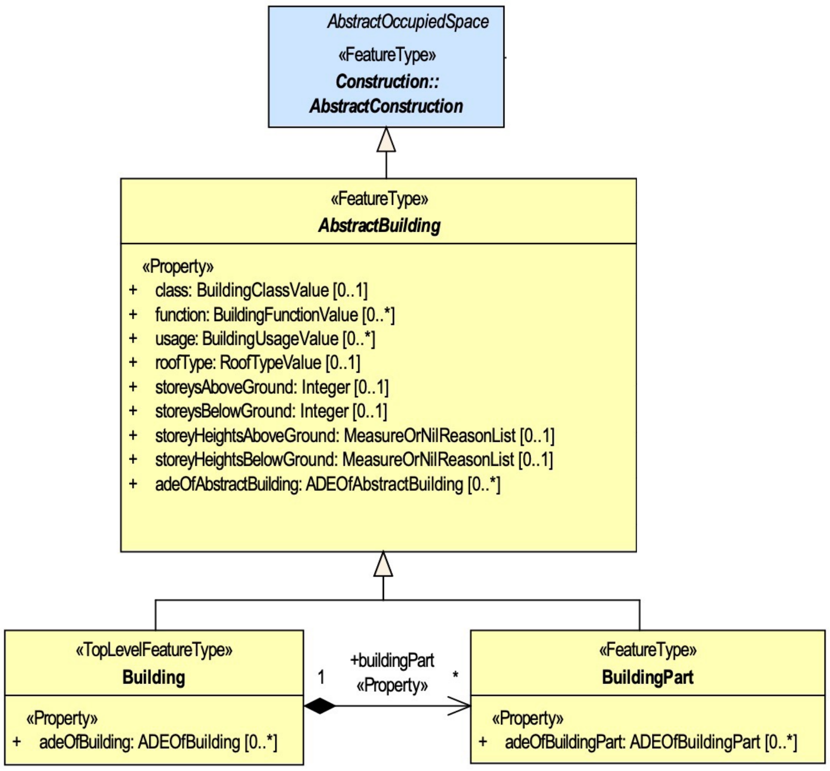

AbstractOccupiedSpace
«FeatureType»
Construction::
AbstractConstruction
«FeatureType»
AbstractBuilding
«Property»
+ class: BuildingClassValue [0..1]
+ function: BuildingFunctionValue [0..*]
+ usage: BuildingUsageValue [0..*]
+ roofType: RoofTypeValue [0..1]
+ storeysAboveGround: Integer [0..1]
+ storeysBelowGround: Integer [0..1]
+ storeyHeightsAboveGround: MeasureOrNilReasonList [0..1]
+ storeyHeightsBelowGround: MeasureOrNilReasonList [0..1]
+ adeOfAbstractBuilding: ADEOfAbstractBuilding [0..*]
«TopLevelFeatureType»
Building
«Property»
+ adeOfBuilding: ADEOfBuilding [0..*]
1
+buildingPart
«Property»
*
«FeatureType»
BuildingPart
«Property»
+ adeOfBuildingPart: ADEOfBuildingPart [0..*]

Figure 2.8.: CityGML v.3.0 Package UML diagram - Building module (excerpt)
(Figure adapted from [20])

As this research aims to support the new 3DCityDB developed for the CityGML v.3.0 standards, differentiating the changes in geometry representations is essential. The new Space concepts provide a clear scope for collecting all possible feature geometry representations as the entries of GIS layers.

2.2. 3D City Database

The 3DCityDB up to version 4.x is an open-source geo-database suite allowing the import, management, analysis, visualisation, and export of virtual 3D city models according to the CityGML standards, supporting both versions 1.0 and 2.0. This data encoding type is based on the Spatially-extended Relational Database Management System (SDBMS), which supports all required geometry types and provides proper spatial indexing for both geometric and topological analyses. It is compatible with most GIS tools [13]. SDBMS is usually deployed on the servers to handle heavy data transactions made by the access to large 3D spatial data files like GML and JSON [19].

The 3DCityDB schema is established to accommodate the CityGML model for storage and processing. However, optimisations are applied for the database design to convert the conceptual model into compacted relational tables; otherwise, a one-to-one mapping of the CityGML data model will result in a vast number of tables and relations in between [19].

This section gives a brief introduction to the database schema of both 3DCityDB v.4.x and 3DCityDB v.5.0, including the data schema tables that are frequently referenced by SQL queries for feature geometry and attribute retrieval.

2.2.1. 3DCityDB v.4.x

The latest release of 3DCityDB v.4.4.0 maps classes in CityGML v.2.0 to 66 database relational tables in a PostgreSQL database schema, generating a default schema named "citydb" when installed. It also allows creating multiple schemas with different names to accommodate data regarding different scenarios. Additionally, a package of PL/pgSQL functions named "citydb_pkg" is installed when the default schema is set up. The functions package offers frequently used procedures [29] like the delete feature and cleaning the schema to facilitate the use of 3DCityDB.

After a successful 3DCityDB set-up, spatial data can be imported into the database from a CityGML or CityJSON file with the "3D City Database Importer/Exporter" tool. Users can then access and use the data via SQL queries. 3DCityDB provides customised data import, which allows users to import features within a specific extent or filter attributes based on specified conditions. Moreover, a database report is made available for users to overview the existing data in the 3DCityDB. Finally, the 3DCityDB supports the stored data to be exported into files in the format of GML, Comma-Separated Values (CSV), XML, JSON, KML, COLLADA, and glTF.

The 3DCityDB v.4.x encodes the city object classes in CityGML data concerning the database complexity, operating performance and semantic interoperability. According to the mapping characteristics from [13], the abstract class that holds all attributes and associations will be inherited by the concrete sub-classes, and each of the sub-classes shall not have any further attributes or be associated with other classes. The demonstration of converting the conceptual model of the CityGML data model to the relational tables of the Building class in CityGML v.2.0 is shown in Figure 2.9 as an example.

The rules above are applied by 3DCityDB to map classes belonging to an inheritance hierarchy onto one table. For instance, the CITYOBJECT table in the 3DCityDB v.4.x database schema stores all features and attributes derived from the starting point abstract class in the Core module, the "_CityObject" classes. For the abstract classes in different thematic modules (e.g. Building module), a separate table associated with the CITYOBJECT table is created (e.g. BUILDING table) to accommodate multiple sub-classes features (e.g. buildings, building parts) and all the shared attributes since they are derived from the same abstract thematic class, i.e. "_AbstractBuilding" class. Moreover, an additional column `objectclass_id` is added to distinguish different classes of the features stored in the abstract class tables [13] (Figure 2.10). There are other tables in 3DCityDB v.4.x, such as the CITYOBJECT_GENERICATTRIB table, to store the "_genericAttribute" data type information related to all sub-class features derived from the "_CityObject" class (see Figure 2.1). Additionally, there is the ADE table for storing the meta-information of an ADE, e.g., its name, description, version, etc. ADE information is stored in separate data tables in 3DCityDB v.4.x. However, it is outside the scope of this research.

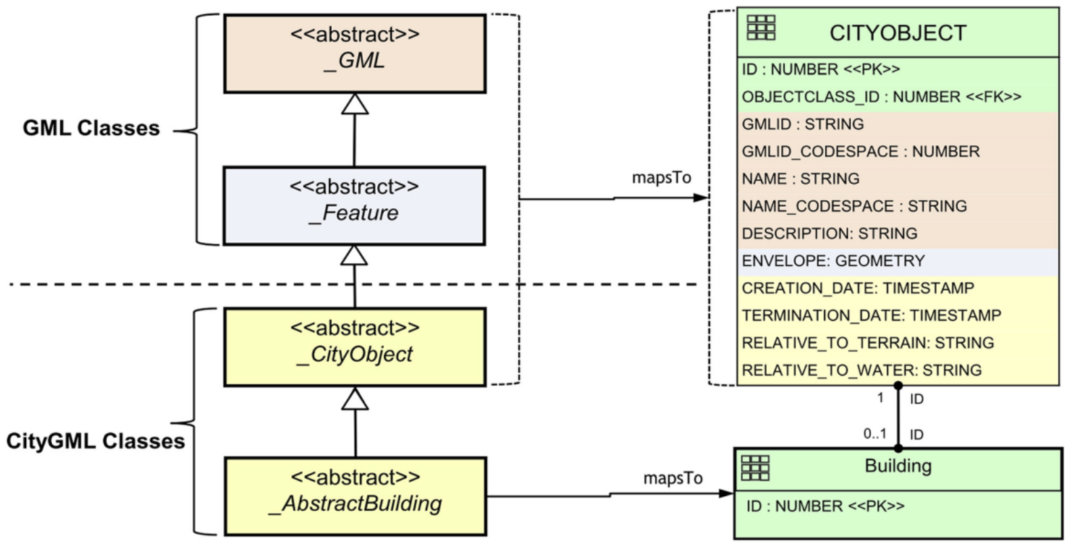


Figure 2.9.: 3DCityDB Inheritance of "Building" class features mapping in CityGML v.2.0 (Figure from [13])

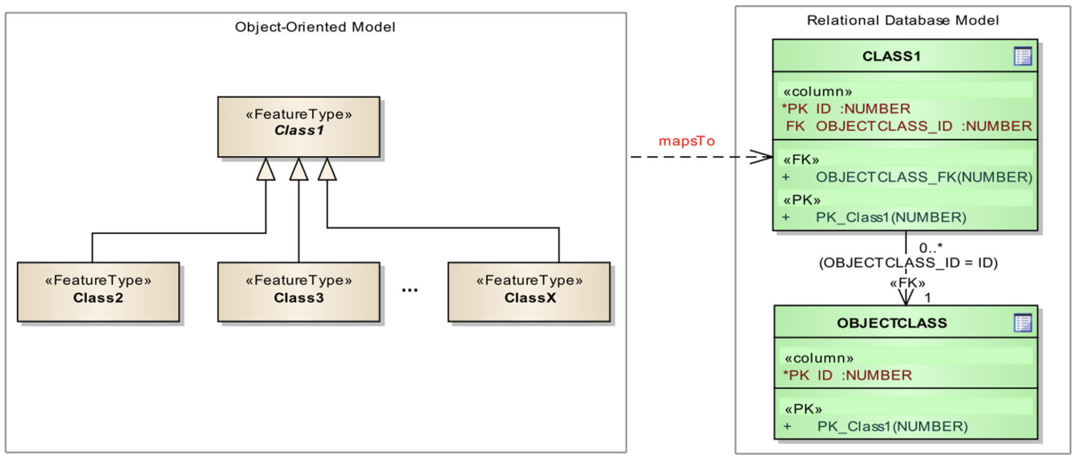
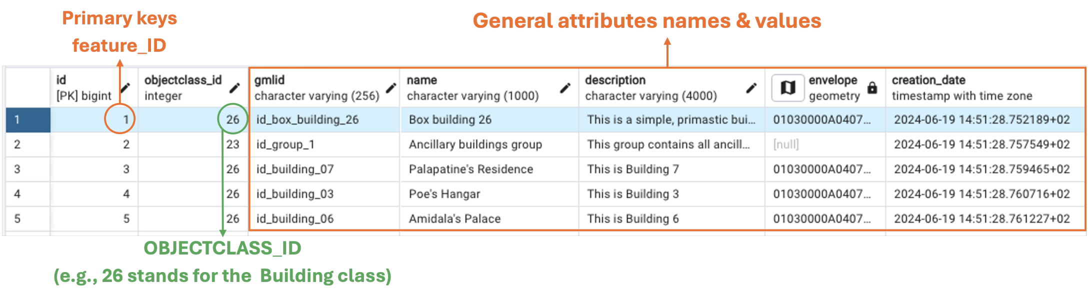


Figure 2.10.: Example of mapping multiple classes onto one table in 3DCityDB (Figure from [13])

In this approach, the sub-classes are logically mapped onto one table in the sat the same inheritance hierarchy level to avoid multiple joins when retrieving data from-classes. It also retains the storage efficiency as only the `objectclass_id` indicating the class names is required to be added to the table [13]. The commonly used tables and their columns in the 3DCityDB v.4.x for data usage are briefly introduced in this paragraph, which include:

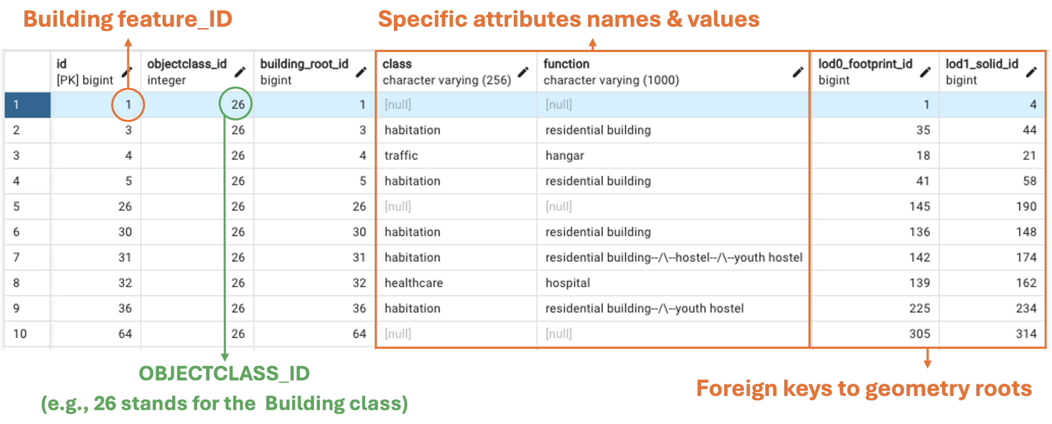
- CITYOBJECT table: It is used as a general registry table for querying the desired features, containing the ID values of each feature as primary keys for referencing. The `objectclass_id` values indicate the specific class names. General attributes like `gmlid` and `name` are inherited by the thematic features from the super abstract class "_CityObject". These general attributes are flattened (linearised) and associated with each unique feature. The `envelope` refers to the bounding box geometry of features, which can be used for extent selection at the data import (Figure 2.11).


Primary keys
feature_ID
General attributes names & values
OBJECTCLASS_ID
(e.g., 26 stands for the Building class)

	id [PK] bigint	objectclass_id integer	gmlid character varying (256)	name character varying (1000)	description character varying (4000)	envelope geometry	creation_date timestamp with time zone
1	1	26	id_box_building_26	Box building 26	This is a simple, primastic bui...	01030000A0407...	2024-06-19 14:51:28.752189+02
2	2	23	id_group_1	Ancillary buildings group	This group contains all ancill...	[null]	2024-06-19 14:51:28.757549+02
3	3	26	id_building_07	Palapatine's Residence	This is Building 7	01030000A0407...	2024-06-19 14:51:28.759465+02
4	4	26	id_building_03	Poe's Hangar	This is Building 3	01030000A0407...	2024-06-19 14:51:28.760716+02
5	5	26	id_building_06	Amidala's Palace	This is Building 6	01030000A0407...	2024-06-19 14:51:28.761227+02

Figure 2.11.: Example of the CITYOBJECT table in 3DCityDB v.4.x

- "LINKED" tables: The "LINKED" table here is not an exact schema table name in the 3DCityDB v.4.x but is used to refer to the abstract class tables from the thematic modules. It contains primary ID keys and `objectclass_ids` that refer to unique thematic features and their specific attributes stored in the feature-based approach, which are already flattened (linearised). The foreign keys to the geometry roots are included according to the geometry and LoDs representations specifications in different thematic modules. Examples of "LINKED" tables can be the BUILDING table (Figure 2.12) for storing features derived from the "_AbstractBuilding" class such as "Building" and "BuildingPart".


Building feature_ID
Specific attributes names & values
OBJECTCLASS_ID
(e.g., 26 stands for the Building class)
Foreign keys to geometry roots

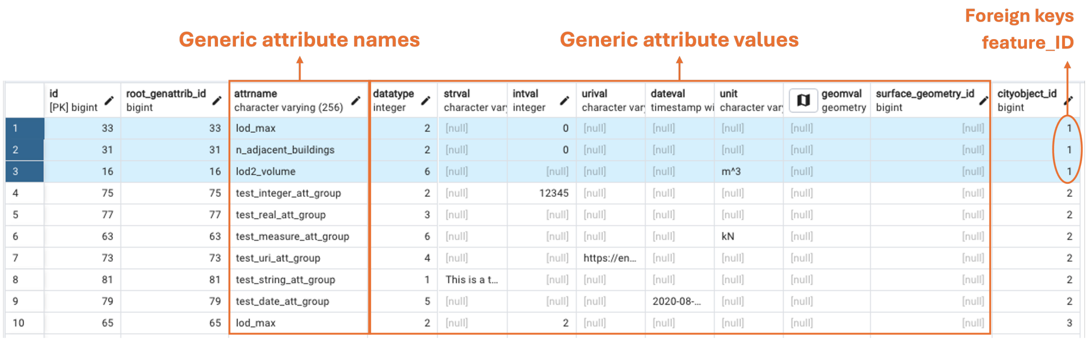
	id [PK] bigint	objectclass_id integer	building_root_id bigint	class character varying (256)	function character varying (1000)	lod0_footprint_id bigint	lod1_solid_id bigint
1	1	26	1	[null]	[null]	1	4
2	3	26	3	habitation	residential building	35	44
3	4	26	4	traffic	hangar	18	21
4	5	26	5	habitation	residential building	41	58
5	26	26	26	[null]	[null]	145	190
6	30	26	30	habitation	residential building	136	148
7	31	26	31	habitation	residential building--/\--hostel--/\--youth hostel	142	174
8	32	26	32	healthcare	hospital	139	162
9	36	26	36	habitation	residential building--/\--youth hostel	225	234
10	64	26	64	[null]	[null]	305	314

Figure 2.12.: Example of the BUILDING table in 3DCityDB v.4.x

- CITYOBJECT_GENERICATTRIB table: It is used to store all generic attributes of the thematic features, deriving from the "_CityObject", the starting point abstract class in the Core module. Unlike the BUILDING table, where attributes are stored inline and associated with unique `feature_ids` (SFS model), the generic attributes of buildings are stored following the Entity-Attribute-Value Model (EAV model). The EAV model organises data into three columns: Entity, Attribute, and Value. In this model, an entity represents an abstract object of some sort, and a collection of similar entities forms an entity set which resembles a class of objects (e.g., buildings). An attribute refers to a specific property of the entities in that set (e.g., building colour), and the value stores the actual data associated with the attribute (e.g., "yellow") [30].

The EAV model is designed to handle situations where an entity could potentially have a huge number of attributes, but these attributes are not pre-defined or fixed in the schema. Instead, new attributes can be added as needed without modifying the database structure. Due to this flexibility, the EAV model is referred to as an open schema or vertical database, as it supports the dynamic vertical storage of attributes [31].

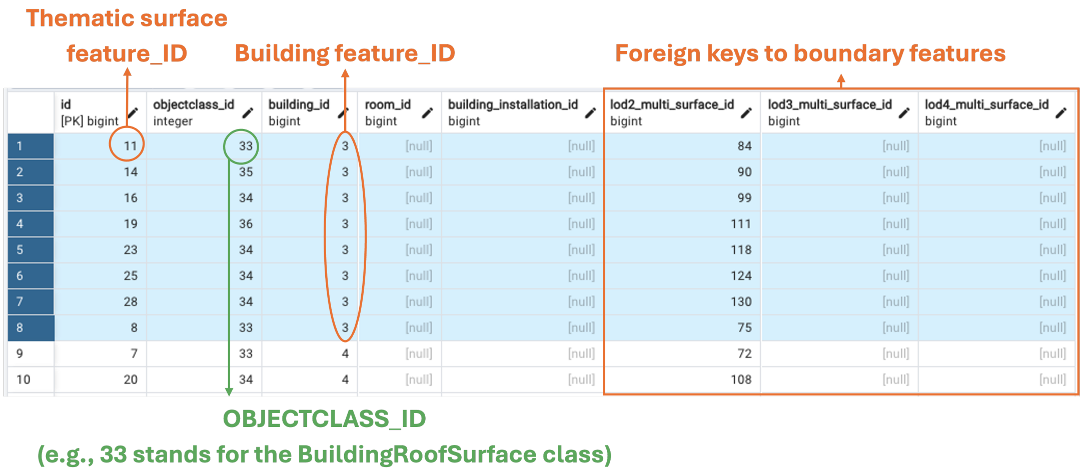
Generic attributes are stored as separate attribute entries with their values in corresponding columns according to their data types. They are associated with the features (entity set) using foreign keys of `feature_ids`. This method allows the generic attributes to be joined with the features as an associated table, displaying all related generic attributes of each feature (Figure 2.13).



	id [PK] bigint	root_genattrib_id bigint	attrname character varying (256)	datatype integer	strval character var	intval integer	urival character var	dateval timestamp wi	unit character var	geomval geometry	surface_geometry_id bigint	cityobject_id bigint
1	33	33	lod_max	2	[null]	0	[null]	[null]	[null]	[null]	[null]	1
2	31	31	n_adjacent_buildings	2	[null]	0	[null]	[null]	[null]	[null]	[null]	1
3	16	16	lod2_volume	6	[null]	[null]	[null]	[null]	m^3	[null]	[null]	1
4	75	75	test_integer_att_group	2	[null]	12345	[null]	[null]	[null]	[null]	[null]	2
5	77	77	test_real_att_group	3	[null]	[null]	[null]	[null]	[null]	[null]	[null]	2
6	63	63	test_measure_att_group	6	[null]	[null]	[null]	[null]	kN	[null]	[null]	2
7	73	73	test_uri_att_group	4	[null]	[null]	https://en...	[null]	[null]	[null]	[null]	2
8	81	81	test_string_att_group	1	This is a t...	[null]	[null]	[null]	[null]	[null]	[null]	2
9	79	79	test_date_att_group	5	[null]	[null]	[null]	2020-08-...	[null]	[null]	[null]	2
10	65	65	lod_max	2	[null]	2	[null]	[null]	[null]	[null]	[null]	3

Figure 2.13.: Example of the CITYOBJECT_GENERICATTRI table in 3DCityDB v.4.x

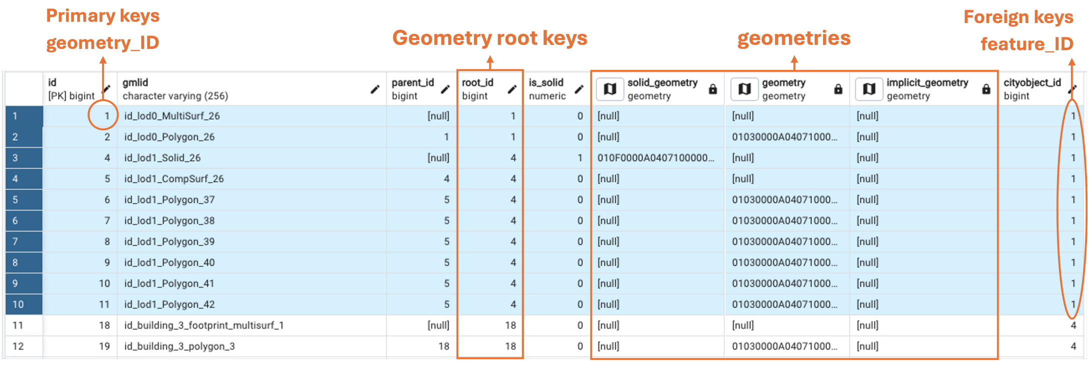
- THEMATIC_SURFACE table: It stores the boundary feature relations, indicated by the `lod(number)_multi_surface_ids`. Relations between features driving from the "_BoundarySurface" abstract class, such as "WallSurface", "GroundSurface", "RoofSurface", and their related "Building" features can then be retained. For example, the building of `feature_id` 3 shown in Figure 2.14 can be represented by its related thematic surfaces, including wall, ground and roof surfaces. Since the LoD concepts are associated with thematic modules, limiting the availability of geometry representations, the thematic surfaces of features like buildings are only available from LoD2 onwards.


Thematic surface feature_ID
Building feature_ID
Foreign keys to boundary features
OBJECTCLASS_ID
(e.g., 33 stands for the BuildingRoofSurface class)

	id [PK] bigint	objectclass_id integer	building_id bigint	room_id bigint	building_installation_id bigint	lod2_multi_surface_id bigint	lod3_multi_surface_id bigint	lod4_multi_surface_id bigint
1	11	33	3	[null]	[null]	84	[null]	[null]
2	14	35	3	[null]	[null]	90	[null]	[null]
3	16	34	3	[null]	[null]	99	[null]	[null]
4	19	36	3	[null]	[null]	111	[null]	[null]
5	23	34	3	[null]	[null]	118	[null]	[null]
6	25	34	3	[null]	[null]	124	[null]	[null]
7	28	34	3	[null]	[null]	130	[null]	[null]
8	8	33	3	[null]	[null]	75	[null]	[null]
9	7	33	4	[null]	[null]	72	[null]	[null]
10	20	34	4	[null]	[null]	108	[null]	[null]

Figure 2.14.: Example of the THEMATIC_SURFACE table in 3DCityDB v.4.x

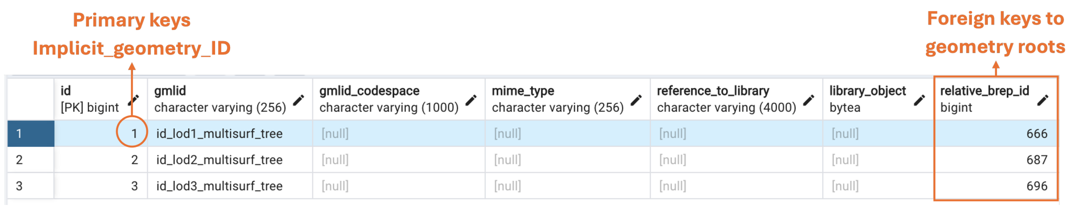
- SURFACE_GEOMETRY table: It stores all the primitive geometries decomposed into planar 3D polygons and solid geometries identified by the primary key `ids`. The `root_ids` are set to specify the geometry roots for aggregation. The geometries stored in it are collected for feature geometry representations when the feature spatial properties are queried (Figure 2.15).


Primary keys geometry_ID
Geometry root keys
geometries
Foreign keys feature_ID

	id [PK] bigint	gmlid character varying (256)	parent_id bigint	root_id bigint	is_solid numeric	solid_geometry geometry	geometry geometry	implicit_geometry geometry	cityobject_id bigint
1	1	id_lod0_MultiSurf_26	[null]	1	0	[null]	[null]	[null]	1
2	2	id_lod0_Polygon_26	1	1	0	[null]	01030000A04071000...	[null]	1
3	4	id_lod1_Solid_26	[null]	4	1	010F0000A0407100000...	[null]	[null]	1
4	5	id_lod1_CompSurf_26	4	4	0	[null]	[null]	[null]	1
5	6	id_lod1_Polygon_37	5	4	0	[null]	01030000A04071000...	[null]	1
6	7	id_lod1_Polygon_38	5	4	0	[null]	01030000A04071000...	[null]	1
7	8	id_lod1_Polygon_39	5	4	0	[null]	01030000A04071000...	[null]	1
8	9	id_lod1_Polygon_40	5	4	0	[null]	01030000A04071000...	[null]	1
9	10	id_lod1_Polygon_41	5	4	0	[null]	01030000A04071000...	[null]	1
10	11	id_lod1_Polygon_42	5	4	0	[null]	01030000A04071000...	[null]	1
11	18	id_building_3_footprint_multisurf_1	[null]	18	0	[null]	[null]	[null]	4
12	19	id_building_3_polygon_3	18	18	0	[null]	01030000A04071000...	[null]	4

Figure 2.15.: Example of the SURFACE_GEOMETRY table in 3DCityDB v.4.x

- IMPLICIT_GEOMETRY table: It is referenced only when features have implicit spatial properties. It stores the `id` primary keys of the implicit geometry attributes and the keys named `relative_brep_id` are used to join with the `root_ids` from the SURFACE_GEOMETRY table, collecting all the geometric components for aggregating the prototypical geometries that are further used for implicit representations (Figure 2.16).



	id [PK] bigint	gmlid character varying (256)	gmlid_codespace character varying (1000)	mime_type character varying (256)	reference_to_library character varying (4000)	library_object bytea	relative_brep_id bigint
1	1	id_lod1_multisurf_tree	[null]	[null]	[null]	[null]	666
2	2	id_lod2_multisurf_tree	[null]	[null]	[null]	[null]	687
3	3	id_lod3_multisurf_tree	[null]	[null]	[null]	[null]	696

Figure 2.16.: Example of the IMPLICIT_GEOMETRY table in 3DCityDB v.4.x

A typical query example used to extract the geometry of features usually involves the tables above. Listing 2.1 gives an example of retrieving all roofs of buildings in LoD2 Multi-Surface geometry representation from the CityGML data stored in the default 3DCityDB v.4.x schema "citydb". The query steps are explained below:

1. THEMATIC_SURFACE is called first to join with the CITYOBJECT table and filtered by the `objectclass_ids` to get all the roofs.
2. BUILDING table is joined to further sort out the roofs that belong to buildings, which is also achieved using the `objectclass_ids`.
3. SURFACE_GEOMETRY table is referenced by joining the `root_ids` with the desired geometry representation and LoD key values, i.e. LoD`2_multi_surface_id` from THEMATIC_SURFACE table to collect all the related polygons for aggregation.
4. The collected geometries are grouped by the `cityobject_id` for aggregation, showing the multi-surfaces of roof features for each building.

```sql
SELECT
   sg.cityobject_id AS co_id,
   st_collect(sg.geometry)::geometry(MultiPolygonZ,28992) AS geom
FROM citydb.thematic_surface AS o
    INNER JOIN citydb.cityobject AS co
       ON o.id = co.id AND o.objectclass_id = 33 -- RoofSurface class
    INNER JOIN citydb.building AS b
       ON o.building_id = b.id AND b.objectclass_id = 26 -- Building class
    INNER JOIN citydb.surface_geometry AS sg
       ON sg.root_id = o.lod2_multi_surface_id AND sg.geometry IS NOT NULL
GROUP BY sg.cityobject_id;
```

Listing 2.1: Example SQL query to extract all roofs of buildings in LoD2 Multi-Surface from the data encoded in 3DCityDB v.4.x

2.2.2. 3DCityDB v.5.0

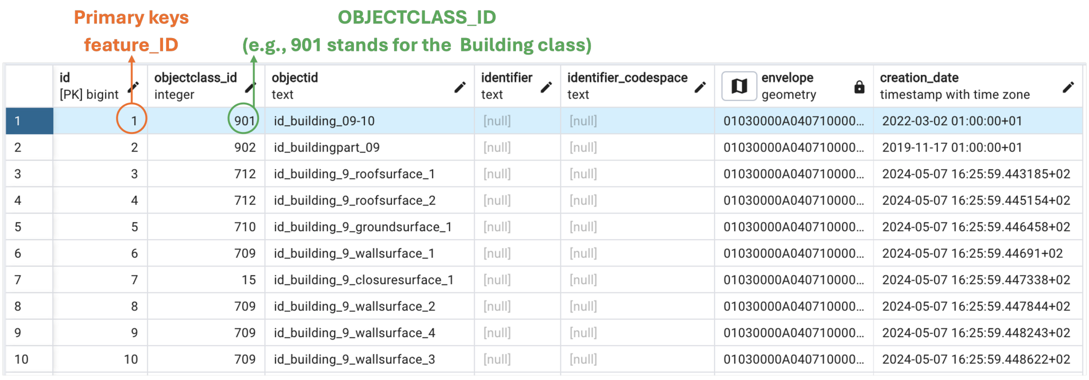
The CityGML v.3.0 introduces fundamental changes (see Section 2.1.2) to the data model. The 3DCityDB of version 0.8.1-beta [22] is used in this research for managing spatial data following CityGML v.3.0 standards. Since the 3DCityDB has been continuously updated, for convenience of reference, the 3DCityDB v.5.0 is used to distinguish all the new versions of the 3DCityDB from the 3DCityDB v.4.x.

The 3DCityDB v.5.0 follows the same mapping rules as the 3DCityDB v.4.x described in Section 2.2.1, in which the abstract class shall hold all the attributes and associations for the inheritance of the concrete subclasses, and each of the subclasses shall not have any further attributes or associated with other classes [13]. However, since the geometry and LoD concepts are elevated to the Core module and are inherited by all thematic classes in CityGML v.3.0 (see Section 2.1.2), the schema tables of thematic abstract class, the "LINKED" tables mentioned in Section 2.2.1 are removed, reducing the schema of the 3DCityDB v.5.0 to 17 tables.

The starting point abstract class table in 3DCityDB v.5.0 is renamed to the FEATURE table, corresponding to the CITYOBJECT table in 3DCityDB v.4.x. All the feature attributes are integrated and stored in the PROPERTY table following the EAV model, which, from the structural point of view, bears similarities with the CITYOBJECT_GENERICATTRIB table in 3DCityDB v.4.x. The PROPERTY table has also been modified to accommodate the attributes of thematic feature classes and the feature boundary relations. The revision of the 3DCityDB schema reduces the operational complexity when performing queries to use the data stored within 3DCityDB v.5.0. However, it also results in certain limitations, which are discussed in Chapter 4.

The commonly referenced tables and their columns in the 3DCityDB v.5.0 for data usage are briefly introduced in this paragraph, which include:

- FEATURE table: It stores all general information of features from the data stored in the 3DCityDB v.5.0. It contains the primary key `ids` as an identifier of features. The `objectclass_id` is included for differentiating the class names. The `objectid`, the former `gmlid` in 3DCityDB v.4.x is stored as another identifier of features. Lastly, the `envelope` is also included for storing the feature bounding box geometries, which can be used for extent selection at the data import (Figure 2.17).



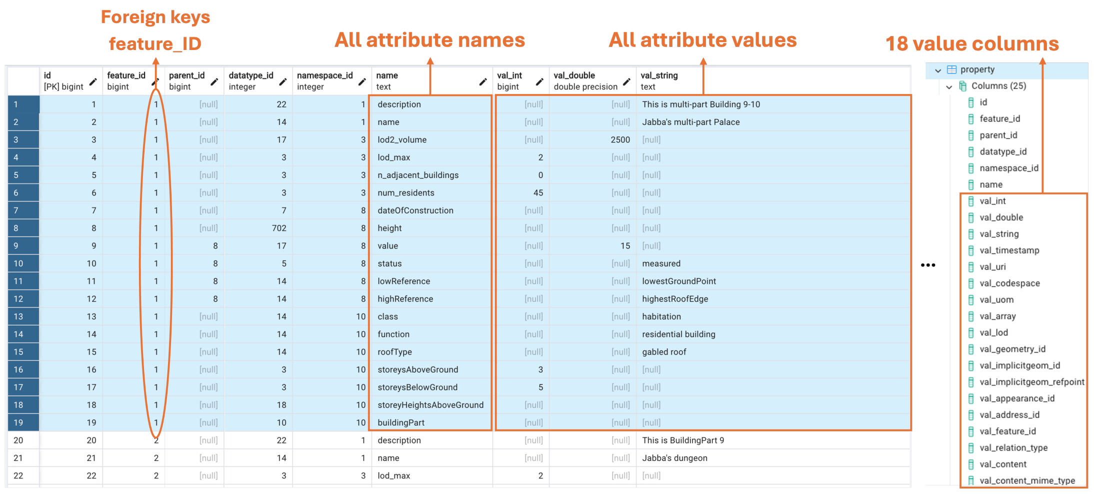
	id [PK] bigint	objectclass_id integer	objectid text	identifier text	identifier_codespace text	envelope geometry	creation_date timestamp with time zone
1	1	901	id_building_09-10	[null]	[null]	01030000A040710000...	2022-03-02 01:00:00+01
2	2	902	id_buildingpart_09	[null]	[null]	01030000A040710000...	2019-11-17 01:00:00+01
3	3	712	id_building_9_roofsurface_1	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.443185+02
4	4	712	id_building_9_roofsurface_2	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.445154+02
5	5	710	id_building_9_groundsurface_1	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.446458+02
6	6	709	id_building_9_wallsurface_1	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.44691+02
7	7	15	id_building_9_closuresurface_1	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.447338+02
8	8	709	id_building_9_wallsurface_2	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.447844+02
9	9	709	id_building_9_wallsurface_4	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.448243+02
10	10	709	id_building_9_wallsurface_3	[null]	[null]	01030000A040710000...	2024-05-07 16:25:59.448622+02

Figure 2.17.: Example of the FEATURE table in 3DCityDB v.5.0

- PROPERTY table: It accommodates all feature attributes and relations following the EAV model. (Figure 2.18). The columns frequently queried from this table are shown as the following:
  - **id**: It is the primary key, an identifier of attribute records.
  - **feature_id**: It is a foreign key to link the attribute to the corresponding features.
  - **parent_id**: It stores the relation of "Nested attributes". These interconnected attributes are termed nested attributes in this research. In the 3DCityDB v.5.0 encoding,

each nested attribute is stored across multiple rows, with parent attributes appearing first, followed by their child attributes linked via `parent_id` keys. Further elaboration and examples of nested attributes will be provided in Chapter 4.

- **datatype_id**: It indicates the available data types in CityGML v.3.0 stored in another schema table named DATATYPE. It is used to differentiate attribute types such as integer, string, feature property, geometry or implicit geometry properties.
- **namespace_id**: It specifies which module of the attributes belong, which can be linked to another schema table named NAMESPACE to select features from certain modules.
- **name**: It stands for the attribute name, which is a vital column for querying the data based on the feature attributes. For example, the spatial properties of building features can have the property like "lod1Solid", specifying buildings represented as a Solid geometry in LoD1.
- **val_**: The column name prefix refers to the value columns storing the attribute values. Based on the attribute data type, values of attributes are stored across different `val_(data_type)` columns, starting from `val_int` to `val_content_mine_type`. There are 18 different value columns in the PROPERTY table, which are detailed in Section 4.3.1.3.

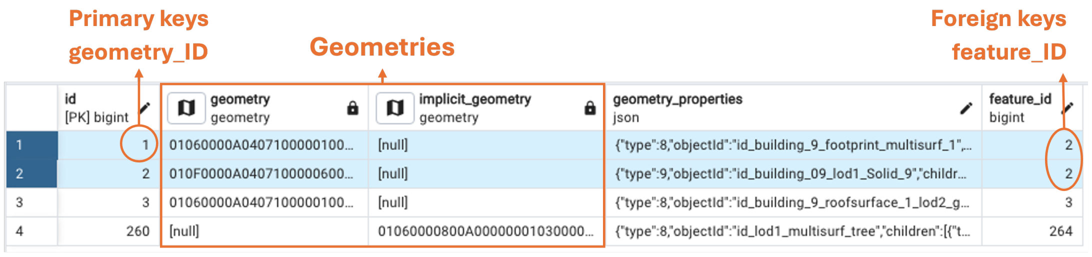


	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	namespace_id integer	name text	val_int bigint	val_double double precision	val_string text
1	1	1	[null]	22	1	description	[null]	[null]	This is multi-part Building 9-10
2	2	1	[null]	14	1	name	[null]	[null]	Jabba's multi-part Palace
3	3	1	[null]	17	3	lod2_volume	[null]	2500	[null]
4	4	1	[null]	3	3	lod_max	2	[null]	[null]
5	5	1	[null]	3	3	n_adjacent_buildings	0	[null]	[null]
6	6	1	[null]	3	3	num_residents	45	[null]	[null]
7	7	1	[null]	7	8	dateOfConstruction	[null]	[null]	[null]
8	8	1	[null]	702	8	height	[null]	[null]	[null]
9	9	1	8	17	8	value	[null]	15	[null]
10	10	1	8	5	8	status	[null]	[null]	measured
11	11	1	8	14	8	lowReference	[null]	[null]	lowestGroundPoint
12	12	1	8	14	8	highReference	[null]	[null]	highestRoofEdge
13	13	1	[null]	14	10	class	[null]	[null]	habitation
14	14	1	[null]	14	10	function	[null]	[null]	residential building
15	15	1	[null]	14	10	roofType	[null]	[null]	gabled roof
16	16	1	[null]	3	10	storeysAboveGround	3	[null]	[null]
17	17	1	[null]	3	10	storeysBelowGround	5	[null]	[null]
18	18	1	[null]	18	10	storeyHeightsAboveGround	[null]	[null]	[null]
19	19	1	[null]	10	10	buildingPart	[null]	[null]	[null]
20	20	2	[null]	22	1	description	[null]	[null]	This is BuildingPart 9
21	21	2	[null]	14	1	name	[null]	[null]	Jabba's dungeon
22	22	2	[null]	3	3	lod_max	2	[null]	[null]

- property
  - Columns (25)
    - id
    - feature_id
    - parent_id
    - datatype_id
    - namespace_id
    - name
    - val_int
    - val_double
    - val_string
    - val_timestamp
    - val_uri
    - val_codespace
    - val_uom
    - val_array
    - val_lod
    - val_geometry_id
    - val_implicitgeom_id
    - val_implicitgeom_refpoint
    - val_appearance_id
    - val_address_id
    - val_feature_id
    - val_relation_type
    - val_content
    - val_content_mime_type

Figure 2.18.: Example of the PROPERTY table in 3DCityDB v.5.0

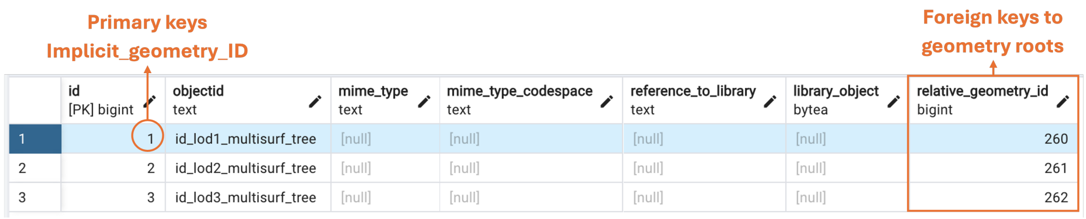
- GEOMETRY_DATA table: It stores all the geometry representations of existing features according to the geometry and LoD concept in CityGML v.3.0 standards. The geometry data in 3DCityDB v.5.0 directly stores the geometries without decomposition following the geometry and LoD changes specified in CityGML v.3.0 (see Section 2.1.2). It contains primary key ID as geometry identifiers to be joined with `val_geometry_ids` or `val_implicitgeom_ids` from the PROPERTY table to collect feature geometries. The `feature_ids` are set as foreign keys to be directly linked with the features (Figure 2.19).



	id [PK] bigint	geometry geometry	implicit_geometry geometry	geometry_properties json	feature_id bigint
1	1	01060000A0407100000100...	[null]	{"type":8,"objectId":"id_building_9_footprint_multisurf_1",...	2
2	2	010F0000A0407100000600...	[null]	{"type":9,"objectId":"id_building_09_lod1_Solid_9","childr...	2
3	3	01060000A0407100000100...	[null]	{"type":8,"objectId":"id_building_9_roofsurface_1_lod2_g...	3
4	260	[null]	01060000800A00000001030000...	{"type":8,"objectId":"id_lod1_multisurf_tree","children":[{"t...	264

Figure 2.19.: Example of the GEOMETRY_DATA table in 3DCityDB v.5.0

- IMPLICIT_GEOMETRY table: It is referenced only when features have implicit spatial properties. It stores the ID primary key of the implicit geometry attributes and the keys named `relative_geometry_ids` are used to join with the IDs from the GEOMETRY_DATA table, retrieving geometries for the feature implicit representations (Figure 2.20).



	id [PK] bigint	objectid text	mime_type text	mime_type_codespace text	reference_to_library text	library_object bytea	relative_geometry_id bigint
1	1	id_lod1_multisurf_tree	[null]	[null]	[null]	[null]	260
2	2	id_lod2_multisurf_tree	[null]	[null]	[null]	[null]	261
3	3	id_lod3_multisurf_tree	[null]	[null]	[null]	[null]	262

Figure 2.20.: Example of the IMPLICIT_GEOMETRY table in 3DCityDB v.5.0

Listing 2.2 shows the same query to retrieve all building roofs represented in LoD2 "MultiSurface" from the CityGML data stored in the default 3DCityDB v.5.0 schema "citydb". It reflects the data model changes in the CityGML v.3.0 where features' LoDs and geometry representations become properties derived from the abstract class in the Core module. The integrated characteristics of 3DCityDB v.5.0 encoding involve the renamed starting point abstract class table, the FEATURE table, which stores all existing features within the schema along with their general attributes. Additionally, the PROPERTY table accommodates the specified thematic feature attributes, feature boundary relations, and generic attributes. The query steps are explained below:

1. FEATURE table is first joined with the PROPERTY table to select buildings. The property name is specified as “boundary” since space features such as buildings can have boundary relation properties associated with boundary features like roofs (see Figure 2.4). After the first join, all buildings that have boundary feature relations are filtered.
2. FEATURE and PROPERTY tables are referenced again to select roofs that are represented in ”LoD2 MultiSurface” from the boundary features of buildings.
3. GEOMETRY_DATA table is referenced by joining the ids of feature geometries with geometry property ids, specifically the val_geometry_ids from the second joined PROPERTY table, to select the related geometries of building roofs directly.

```sql
SELECT
    f1.id AS f_id,
    g.geometry::geometry(MultiPolygonZ,28992) AS geom
FROM citydb.feature AS f
    INNER JOIN citydb.property AS p
        ON f.id = p.feature_id
            AND p.name = 'boundary'
            AND f.objectclass_id = 901 --Building class
    INNER JOIN citydb.feature AS f1
        ON f1.id = p.val_feature_id
            AND f1.objectclass_id = 712 --RoofSurface class
    INNER JOIN citydb.property AS p1
        ON f1.id = p1.feature_id
            AND p1.name = 'lod2MultiSurface'
    INNER JOIN geometry_data AS g ON p1.val_geometry_id = g.id;
```

Listing 2.2: Example SQL query to extract all roofs of buildings in LoD2 Multi-Surface from the data encoded in 3DCityDB v.5.0

There are three main differences in the schema between 3DCityDB v.4.x, and 3DCityDB v.5.0:

- **Renamed starting point abstract class table**
  The starting point abstract class table in 3DCityDB v.5.0 is the FEATURE table, which corresponds to the CITYOBJECT table from 3DCityDB v.4.x. The FEATURE table serves as a registry for the existing city objects stored within the 3DCityDB schema.
- **Integrated feature attributes table**
  Since thematic features in CityGML v.3.0 are derived from the starting point abstract class specified in the Space concepts from the Core module, they can only be space or boundary features with limited LoDs and geometry representations. Consequently, spatial properties and boundary feature relations are integrated into a single PROPERTY table. Spatial properties are encoded as geometry and implicit geometry properties, respectively, and are no longer stored across different thematic abstract class tables.

  This integration drops the ”LINKED” tables mentioned in Section 2.2.1, such as the BUILDING, TUNNEL and THEMATIC_SURFACE tables. Furthermore, boundary feature relations are included as feature properties, recording the hierarchical relations between space and boundary features, such as walls and roofs of buildings. Feature attributes in the PROPERTY table are stored following the EAV model in 3DCityDB v.5.0, which mirrors the structure of the CITYOBJECT_GENERICATTRIB table in 3DCityDB v.4.x.

- **Geometries storage**
  The feature geometries are used similarly to those in 3DCityDB v.4.x. However, the main difference is that the geometries are no longer split into polygons; instead, feature geometries are stored directly as points, curves, solids or multi-surfaces in 3DCityDB v.5.0. The aggregation of planar polygons are no longer necessary, as the inherited spatial properties of space features, boundary features or relief features limit possible feature geometry representations.

Understanding the differences in the 3DCityDB schema is fundamental to this research since it significantly affects the SQL queries for accessing the data stored in the new database encoding. The SQL queries implemented by the current plug-in on the server side will be updated to support 3DCityDB v.5.0, achieving the desired outputs without exposing the users with limited RDBMS experience to complex 3DCityDB structure.

2.3. Views and Materialized Views

The view and materialized view of RDBMS are briefly introduced in this section as they are relevant to the QGIS layer-based concept proposed by the current plug-in, which is further described in Section 2.4.1.

Taking PostgreSQL as an example, views refer to named queries stored in the database [32], representing a saved SQL transformation from a set of base tables to a derived table. Views are re-computed each time they are referenced or queried. In contrast, materialized views are cached results of views stored physically in a temporary virtual table [33].

Views can be time-efficient when querying data from a relatively small dataset. However, when dealing with a large dataset like the 3D city model of a city, materialized views could be faster since the indices supported in PostgreSQL can be built on the materialized view to boost the performance when accessed in the database. The query time could be much faster compared to that of a view. The comparison of view and materialized is shown in Table 2.1.

Comparison	**View**	**Materialized View**
Definition	A saved SQL statement representing queried data from the underlying tables	A temporarily stored table that contains the cached result of a view
Re-run when accessed	Yes	No
Support updates to the queried data	Yes	No
Need to be refreshed	No	Yes
Need memory storage	No	Yes
Query time of large dataset	Usually longer	Usually shorter

Table 2.1.: Comparison of the view and materialized view in PostgreSQL

2.4. 3DCityDB-Tools plug-in for QGIS

The 3DCityDB-Tools plug-in for QGIS is developed to "facilitate management and visualisation of data stored in the 3D City Database, which currently supports CityGML v.1.0 and 2.0 " [15]. Although 3DCityDB provides a powerful SDBMSs for accessing massive semantic 3D city datasets, it remains difficult for practitioners who lack a deep knowledge of the CityGML model and the related SDBMSs to interact with the CityGML data encoded in 3DCityDB. Thus, the plug-in proposes a layer-based concept that allows users to connect to local or remote 3DCityDB instances for PostgreSQL/PostGIS and load the data as "classical" layers into QGIS [15]. The users can then interact with the data more intuitively as most AEC practitioners are more familiar with GIS applications like QGIS, considering its prevalence among the heterogeneous and steadily growing community. CityGML data accessed in layers enhances the flexibility in data editing, and the QGIS built-in functions can further extend the use of CityGML data.

This section reveals the structure of the 3DCityDB-Tools plug-in for QGIS, including the server-side part and the client-side part (Figure 2.21) referenced from the introduction of the current plug-in [15]. The focus will be on the server side of the plug-in called "QGIS Package", which must be installed in the 3DCityDB instance. This research aims to provide more details about the working principles of the layer-based concept, reproduce the layer creation process from the 3DCityDB v.5.0 schema, and eventually enhance the capability of the current plug-in to cope with spatial data following the CityGML v.3.0 standards.

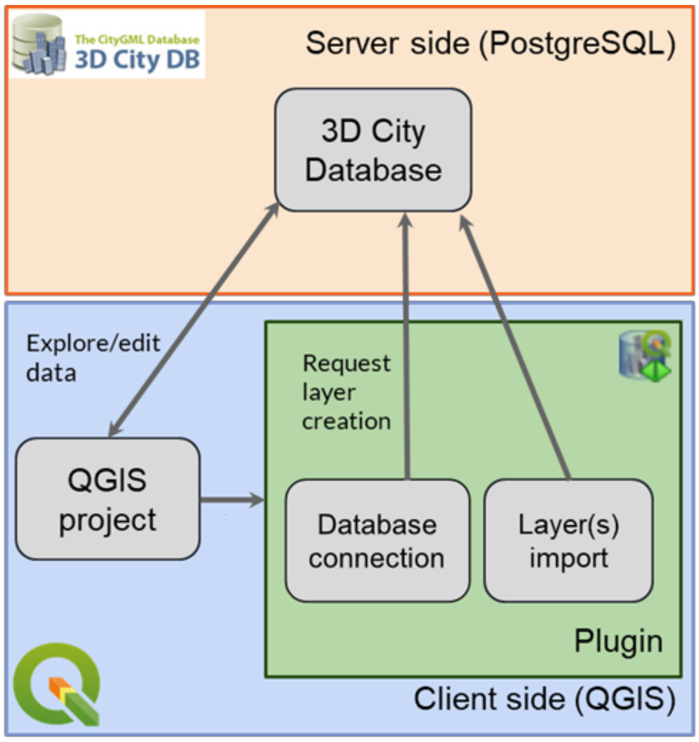


Figure 2.21.: Overall structure of the 3DCityDB-Tools Plug-in for QGIS, with a server-side part for PostgreSQL and a client-side part for QGIS (Figure from [15])

2.4.1. Server-Side

The server-side part of the plug-in, the "QGIS Package", is written in PL/pgSQL. It must be installed on top of a 3DCityDB instance to allow data exchange between 3DCityDB and QGIS. The main functions offered by the "QGIS Package" are targeted at layer creation and the management of users and their privileges to access data in the 3DCityDB instances.

For the layer creation, the "QGIS Package" allows the privileged users to define and create a layer by extracting a specific, selectable geometry according to the LoDs of features together with the corresponding attributes. Each layer consists of a view from the database that links all necessary tables containing the feature attributes and a materialized view containing the feature geometries of the selected LoD [15] geometry type. A simplified example is given by [15] as shown in Figure 2.22. For instance, the attributes of the "Building" class features are stored in tables CITYOBJECT and BUILDING in 3DCityDB v.4.x. These two tables are linked together using the primary/foreign key `id` of building features. The key `id` in the BUILDING table is the identifier of the unique building features, which are then linked to the respective materialized view of feature geometry in a specified LoD geometry type, then the layers are generated using the building geometries as entries complying with the SFS model and become operable in QGIS. For naming conventions, a set of prefixes and suffixes are defined and applied to identify each layer within the same 3DCityDB instances. After layer creation, certain triggers and trigger functions are deployed to make each view updatable (as far as the attributes are concerned).

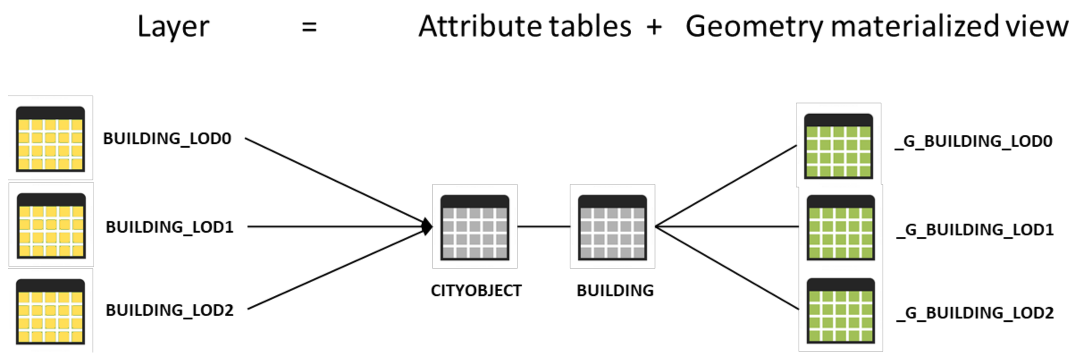


Figure 2.22.: Simplified representation of how layers are composed for the plug-in in the case of "Building" class features(Figure from [15])

The materialized view is chosen for the feature geometries considering the complexity of how geometries are decomposed and stored in the 3DCityDB v.4.x. Its mapping rule is described in Section 2.2.1. All feature geometries are decomposed and stored as 3D planar polygons; the hierarchy and aggregation information are preserved in 3DCityDB v.4.x. It has been proven in the performance test that querying and aggregating the 3D polygons directly from the 3DCityDB v.4.x tables is rather time-consuming [15]. Therefore, the materialized views are chosen as a compromise to provide a better user experience at the cost of sparing storage space and the time needed to generate/refresh them while creating the layers. Additionally, another advantage of using materialized views for feature geometries is that the implicit geometries can be created and stored temporarily in advance for more efficient querying in terms of time. Since the implicit geometries in CityGML are represented using the prototypical geometries that must be instantiated, roto-translated and scaled by 3D affine transformations, running this process during the query is time-consuming.

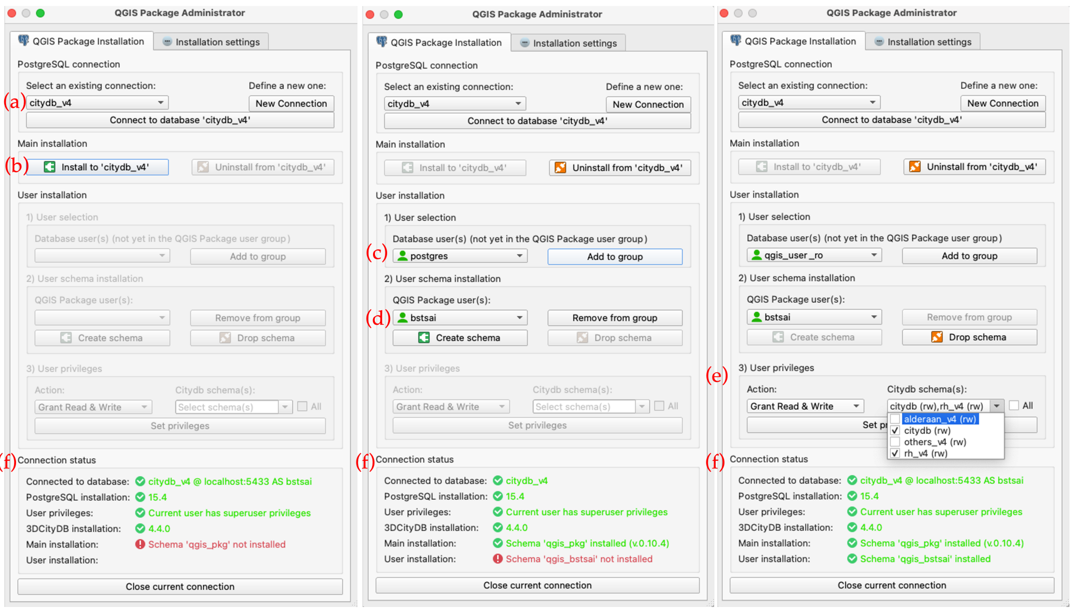
To manage the large number of layers that could result from all possible combinations, several checks have been introduced and implemented. Firstly, layer creation functions are categorised according to CityGML modules (Building, Bridge, Vegetation, Transportation, Terrain, etc.), allowing users to invoke them individually. This means that if a user is only interested in building data, only the corresponding layers will be generated. Secondly, during the layer creation process, a check is performed to count the number of existing features for each layer. For instance, if there is no data regarding “Rooms” or “BuildingInstallations” class features in the database, those layers will not be generated. The same principle applies to LoD that layers are only created if the LoD feature geometry representations are available.

Finally, users can specify the area size for which layers will be generated. This is particularly useful for very large city models, as it allows users to create materialized views for only a selected, smaller area rather than the entire city model. This approach not only reduces the storage space required by the materialized views but also significantly decreases the time needed to refresh them.

2.4.2. Client-Side

The client-side part of the plug-in enables users to interact with the ”QGIS Package” on the server through a set of GUI-based dialogs, and the data in 3DCityDB v.4.x directly within QGIS. The latest version is plug-in version 0.8.x introduced by [15], the client-side part offers three GUI-based tools:

- The **“QGIS Package Administrator”** installs the server-side part of the plug-in, as well as to set up database user access and user privileges. It can only be used by the database administrator. In Figure 2.23, certain functions of the “QGIS Package Administrator” GUI dialog are introduced and indicated by letters. The users first set up the connection to the PostgreSQL database (a), and then the administrator can perform different operations. The “QGIS Package” (`qgis_pkg` for short) must be installed for the server-side part for the first time (b). Once the server-side part is installed, the user installation part is activated, enabling users to choose which database users can connect to the selected 3DCityDB database instance from the plug-in. The user is added to the specific database user group to have access (c). For each group member, the database administrator has to set up the server-side configuration by creating the selected user’s schema, which will be named in the form “qgis_(user name)” (”usr_schema” for short) by default containing layers and settings of the selected user (d). The privileges (read-only or read and write) to the database can be granted or revoked for each group user to the existing 3DCityDB schemas(e). Finally, once the database administrator completes the setup, the GUI dialog can be closed, and the client-side part plug-in is enabled to be used by the user based on their privileges. The Connection status console is at the bottom, showing every action information (f).



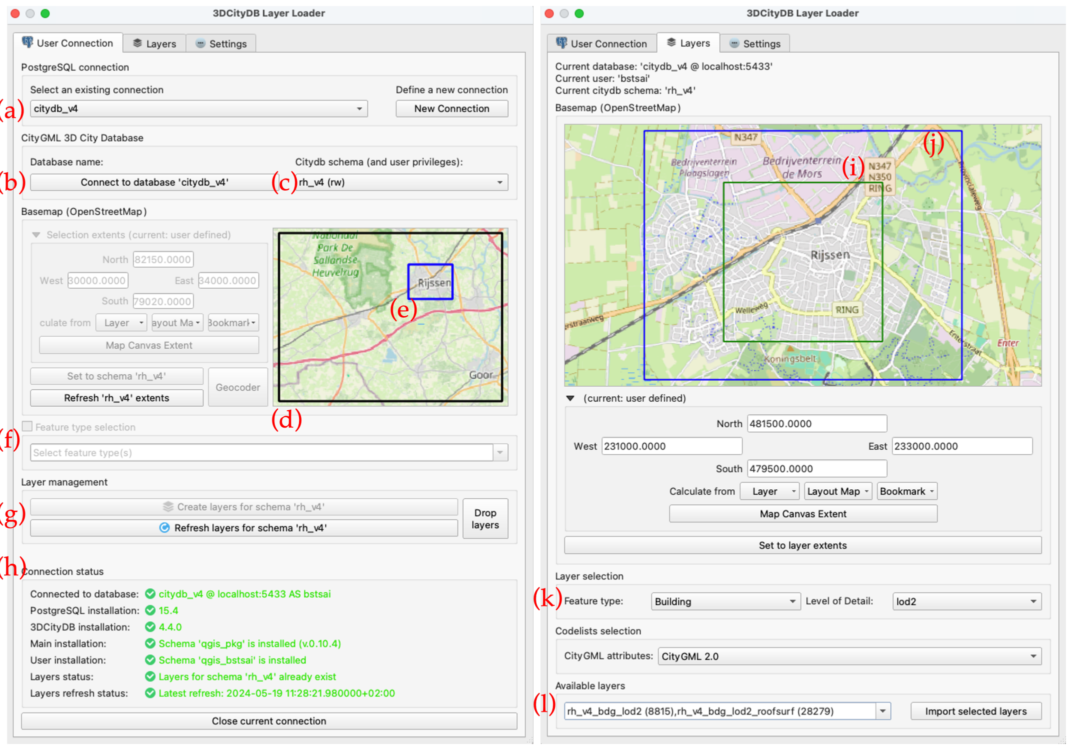
Connected to database: citydb_v4 @ localhost:5433 AS bstsai
PostgreSQL installation: 15.4
User privileges: Current user has superuser privileges
3DCityDB installation: 4.4.0
Main installation: Schema 'qgis_pkg' not installed
User installation:

Connected to database: citydb_v4
PostgreSQL installation: 15.4
User privileges: Current user has superuser privileges
3DCityDB installation: 4.4.0
Main installation: Schema 'qgis_pkg' installed (v.0.10.4)
User installation: Schema 'qgis_bstsai' not installed

Connected to database: citydb_v4 @ localhost:5433 AS bstsai
PostgreSQL installation: 15.4
User privileges: Current user has superuser privileges
3DCityDB installation: 4.4.0
Main installation: Schema 'qgis_pkg' installed (v.0.10.4)
User installation: Schema 'qgis_bstsai' installed

Figure 2.23.: QGIS Package Administrator (v.0.8.9) operations overview

- The **"Layer Loader"** allows users to import and interact with layers in the 3DCityDB v.4.x directly within QGIS, its GUI dialog can be loaded by any user. The functions offered by the Layer Loader are described in Figure 2.24. Once the user connects to a 3DCityDB instance (a,b), the list of available schemas (or "scenarios") is shown, giving the access privileges information (read-only: ro, or read and write: rw) (c). The user can select the 3DCityDB schemas to work with, and the extent of the whole dataset by default is first displayed in the map canvas by the black box, indicating the current extent (d). Users can specify the extent to which the layers will be generated according to their needs. The specified extent will be displayed in the blue box (e). In addition, the user can choose whether feature layers have to be created for all CityGML modules or only the specified part (e.g. Building, Vegetation modules) (f). After the user specifies it, feature layers can be created. Materialized views and updatable views are needed for setting up each layer, and the materialized views can be subsequently refreshed (g). If all the requirements are satisfied, indicated by the Connection status console (h), the user can move to the Layers tab to load layers into QGIS.
  In the Layers tab, users are allowed to specify the extent of data loading again. The newly specified extent is indicated by the green box (i), or by default, users can import all data in the layer extent defined before when in the User connection tab, which is the blue box (j). The layers to be imported are selected and filtered by specifying the CityGML module (e.g. Building) and the LoDs (k). The toggle list of the available layers will be updated, showing only the existing layers in the green bounding box (l).



3DCityDB Layer Loader

User Connection | Layers | Settings

PostgreSQL connection
Select an existing connection: citydb_v4
Define a new connection: New Connection

CityGML 3D City Database
Database name: Connect to database 'citydb_v4'
Citydb schema (and user privileges): rh_v4 (rw)

Basemap (OpenStreetMap)
Selection extents (current: user defined)
North 482150.0000
West 230000.0000 East 234000.0000
South 479020.0000
Calculate from: Layer | Layout Map | Bookmark
Map Canvas Extent
Set to schema 'rh_v4'
Refresh 'rh_v4' extents
Geocoder

Feature type selection
Select feature type(s)

Layer management
Create layers for schema 'rh_v4'
Refresh layers for schema 'rh_v4'
Drop layers

Connection status
Connected to database: citydb_v4 @ localhost:5433 AS bstsai
PostgreSQL installation: 15.4
3DCityDB installation: 4.4.0
Main installation: Schema 'qgis_pkg' is installed (v.0.10.4)
User installation: Schema 'qgis_bstsai' is installed
Layers status: Layers for schema 'rh_v4' already exist
Layers refresh status: Latest refresh: 2024-05-19 11:28:21.980000+02:00
Close current connection

3DCityDB Layer Loader

User Connection | Layers | Settings

Current database: 'citydb_v4 @ localhost:5433'
Current user: 'bstsai'
Current citydb schema: 'rh_v4'
Basemap (OpenStreetMap)
(current: user defined)
North 481500.0000
West 231000.0000 East 233000.0000
South 479500.0000
Calculate from: Layer | Layout Map | Bookmark
Map Canvas Extent
Set to layer extents

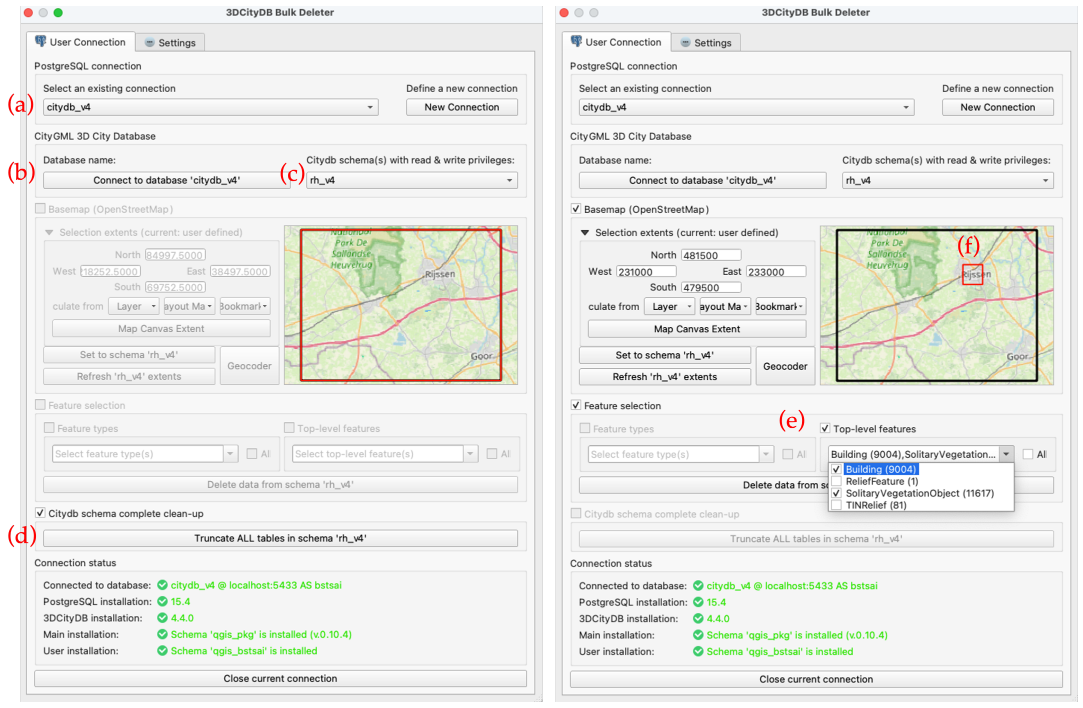
Layer selection
Feature type: Building
Level of Detail: lod2

Codelists selection
CityGML attributes: CityGML 2.0

Available layers
rh_v4_bdg_lod2 (8815),rh_v4_bdg_lod2_roofsurf (28279)
Import selected layers

Figure 2.24.: 3DCityDB Layer Loader (v.0.8.9) operations overview

- The **"Bulk Deleter"** can be used to bulk delete features from the 3DCityDB v.4.x, either all at once or using spatial and feature-related filters, which can only be used by users with read-and-write privileges. Similar to the Layer loader as shown in Figure 2.25, once connected to a 3DCityDB instance (a,b), the list of available schemas (or "scenario") for which read-and-write privileges are granted is shown. There are two ways to delete features once users specify the schema (c). One is to clean up the whole schema, truncating all 3DCityDB schema tables (d). The other is by selecting feature types to delete, choosing either from a list of available CityGML modules or from a list of available CityGML top-level features (e). Likely, it is also possible for users to set the extent of feature deletion; the extent is shown by the red box (f), and the information in the toggle lists for user specification will be updated dynamically according to existing features from the extent selection.



3DCityDB Bulk Deleter

User Connection | Settings

PostgreSQL connection
Select an existing connection: citydb_v4
Define a new connection: New Connection

CityGML 3D City Database
Database name: Connect to database 'citydb_v4'
Citydb schema(s) with read & write privileges: rh_v4

Basemap (OpenStreetMap)
Selection extents (current: user defined)
North 84997.5000
West 18252.5000 East 38497.5000
South 69752.5000
culate from Layer ayout Ma Bookmark
Map Canvas Extent
Set to schema 'rh_v4'
Refresh 'rh_v4' extents
Geocoder

Feature selection
Feature types: Select feature type(s) — All
Top-level features: Select top-level feature(s) — All
Delete data from schema 'rh_v4'

✓ Citydb schema complete clean-up
Truncate ALL tables in schema 'rh_v4'

Connection status
Connected to database: citydb_v4 @ localhost:5433 AS bstsai
PostgreSQL installation: 15.4
3DCityDB installation: 4.4.0
Main installation: Schema 'qgis_pkg' is installed (v.0.10.4)
User installation: Schema 'qgis_bstsai' is installed

Close current connection

3DCityDB Bulk Deleter

User Connection | Settings

PostgreSQL connection
Select an existing connection: citydb_v4
Define a new connection: New Connection

CityGML 3D City Database
Database name: Connect to database 'citydb_v4'
Citydb schema(s) with read & write privileges: rh_v4

✓ Basemap (OpenStreetMap)
Selection extents (current: user defined)
North 481500
West 231000 East 233000
South 479500
culate from Layer ayout Ma Bookmark
Map Canvas Extent
Set to schema 'rh_v4'
Refresh 'rh_v4' extents
Geocoder

✓ Feature selection
Feature types: Select feature type(s) — All
✓ Top-level features: Building (9004),SolitaryVegetation... — All
✓ Building (9004)
ReliefFeature (1)
✓ SolitaryVegetationObject (11617)
TINRelief (81)
Delete data from s

Citydb schema complete clean-up
Truncate ALL tables in schema 'rh_v4'

Connection status
Connected to database: citydb_v4 @ localhost:5433 AS bstsai
PostgreSQL installation: 15.4
3DCityDB installation: 4.4.0
Main installation: Schema 'qgis_pkg' is installed (v.0.10.4)
User installation: Schema 'qgis_bstsai' is installed

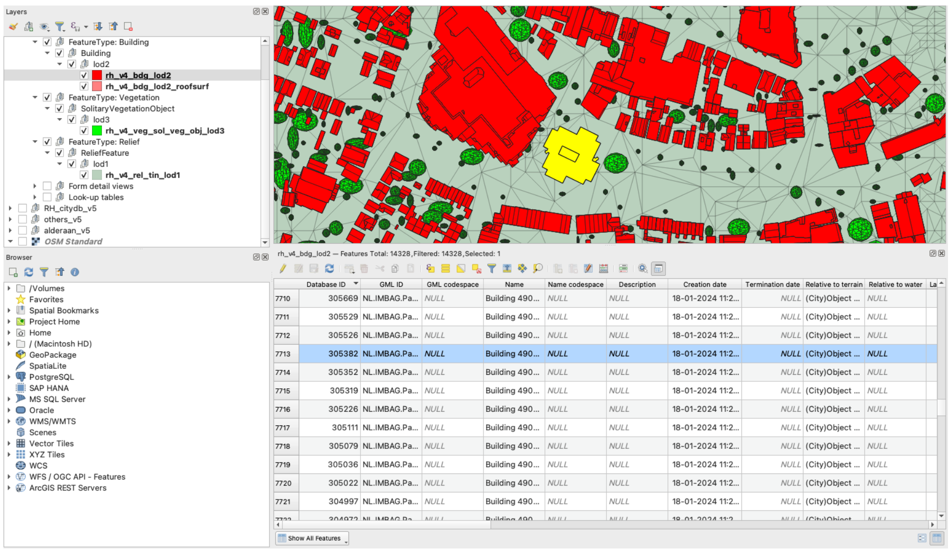
Close current connection

Figure 2.25.: 3DCityDB Bulk Deleter (v.0.8.9) operations overview

2.4.3. Plug-in Test Results and Current Limitations

Once the layers are imported into the QGIS main window, users can interact with them as "normal" GIS layers and perform the usual operations. A hierarchical Table of Content (ToC) (Layers tab) is generated and updated with each layer import, providing an overview of the loaded layers organised by the CityGML modules and LoD. An example is shown in Figure 2.26. Users can select features and access their attributes via the table view or customised attribute forms. These forms display the feature's attributes and include nested tables of CityGML generic attributes, addresses, external references, etc. If users are permitted to edit data, the attribute forms conduct various checks to avoid errors during data entry, and users are visually notified in case of invalid input. For example, error messages are shown in Figure 2.27 due to the false entry of storeys below ground as the value entered must be a positive integer. The other error is caused by missing the unit measuring the height value.

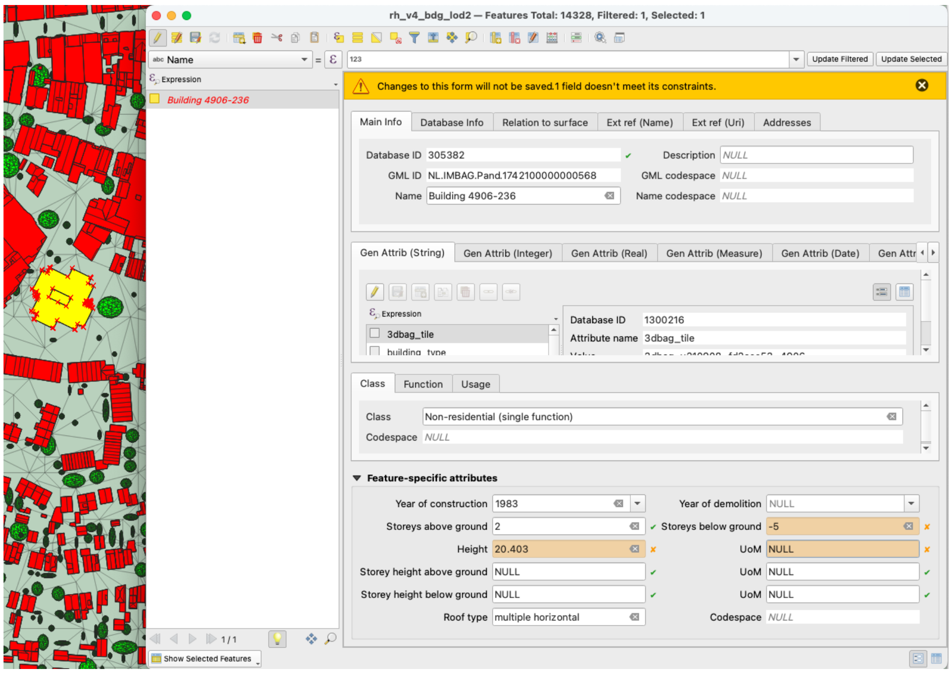
According to [15], the plug-in has been successfully tested with several datasets and earned certain positive feedback from the beta testers. However, several limitations apply to the current plug-in. Firstly, since raster-based layers are not supported by 3DCityDB, they are also not supported by the current plug-in, specifically applying to the "RasterRelief" class. Secondly, the "CityObjectGroup" class is not supported by the "Layer Loader" but is supported by the "Bulk Deleter". Thirdly, no CityGML appearances are supported, which implies that neither colours nor textures can be read from the 3DCityDB v.4.x and applied to the loaded features. Finally, the CityGML ADEs is not supported currently. Despite the limitations, the current plug-in still contributes to bridging the gap between common GIS practitioners and complex 3D city models. As the 3DCityDB v.5.0 has been updated to support CityGML v.3.0, it will be valuable to investigate and integrate support for the current plug-in. Therefore, this research delves into the investigation to reproduce the layer creation process achieved by the server-side “QGIS Package”, providing possible solutions to create loadable layers concerning the data stored in the 3DCityDB v.5.0.



rh_v4_bdg_lod2 — Features Total: 14328, Filtered: 14328, Selected: 1

	Database ID	GML ID	GML codespace	Name	Name codespace	Description	Creation date	Termination date	Relative to terrain	Relative to water
7710	305669	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7711	305529	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7712	305526	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7713	305382	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7714	305352	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7715	305319	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7716	305226	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7717	305111	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7718	305079	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7719	305036	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7720	305022	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL
7721	304997	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:2...	NULL	(City)Object ...	NULL

Figure 2.26.: Example of standard attribute table view of the layers loaded in QGIS



rh_v4_bdg_lod2 — Features Total: 14328, Filtered: 1, Selected: 1

Building 4906-236

Changes to this form will not be saved.1 field doesn't meet its constraints.

Database ID	305382	Description	NULL
GML ID	NL.IMBAG.Pand.1742100000000568	GML codespace	NULL
Name	Building 4906-236	Name codespace	NULL

3dbag_tile
building_type

Database ID	1300216
Attribute name	3dbag_tile

Class	Non-residential (single function)
Codespace	NULL

Feature-specific attributes

Year of construction	1983	Year of demolition	NULL
Storeys above ground	2 ✓	Storeys below ground	-5 ✗
Height	20.403 ✗	UoM	NULL ✗
Storey height above ground	NULL ✓	UoM	NULL ✓
Storey height below ground	NULL ✓	UoM	NULL ✓
Roof type	multiple horizontal	Codespace	NULL

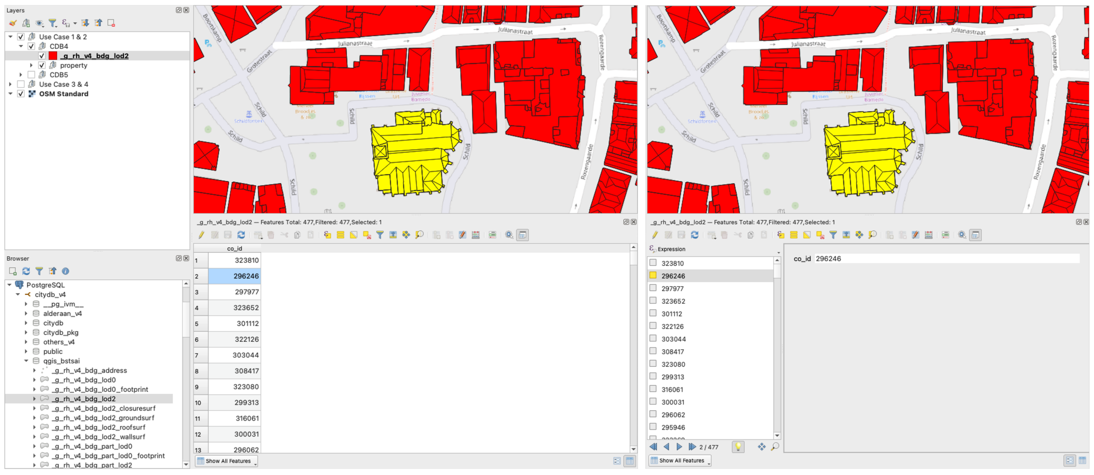
Figure 2.27.: Example of customised attribute form view and the error checking of the layers loaded in QGIS

3. Possible Use Cases Analysis of CityGML Data in QGIS

Before introducing the methodology, it is essential to identify the potential use cases for how users interact with spatial data. This chapter uses the example of CityGML data stored in 3DCityDB through QGIS. Identifying possible spatial data use cases clarifies the research objectives, which aim to enhance server-side support to the 3DCityDB-Tools plug-in for QGIS (plug-in). Four use cases are articulated in this chapter, ranging from accessing feature geometries to interacting with geometries and attributes via QGIS GUIs. Each use case is detailed separately below.

3.1. Case 1: Users Interact Only with Feature Geometries

Inspecting the feature geometries is the basic use of the CityGML data. As introduced in Section 2.3 and Section 2.4, the SQL statements for querying feature geometries can be saved as views or materialized views. After setting up the PostgreSQL database connection, users can drag and drop these available geometry views and load them as GIS layers. Each feature geometry with its unique `cityobject_id(co_id)` can then be inspected in 2D (Figure 3.1). Users can visualise feature geometries in 3D using either the QGIS 3D Map or the plug-in called "Qgis2threejs" (Figure 3.2).



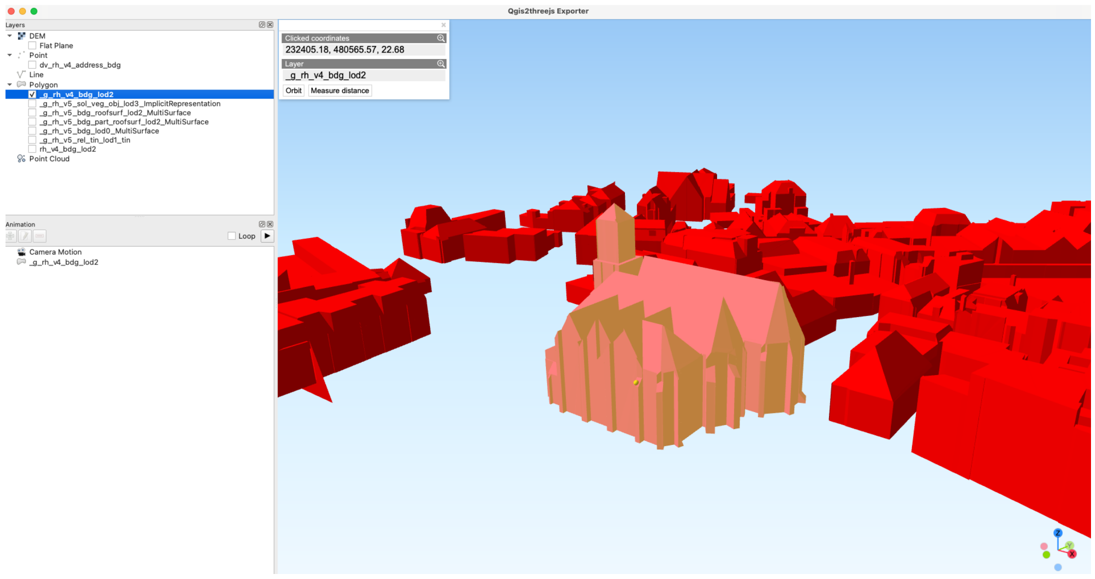
_g_rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 1

	co_id
1	323810
2	296246
3	297977
4	323652
5	301112
6	322126
7	303044
8	308417
9	323080
10	299313
11	316061
12	300031
13	296062

_g_rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 1

co_id 296246

Figure 3.1.: Use case 1 example - feature geometries viewed in 2D (Left) attribute table view. (Right) attribute form view


Qgis2threejs Exporter
Layers
DEM
Flat Plane
Point
dv_rh_v4_address_bdg
Line
Polygon
_g_rh_v4_bdg_lod2
_g_rh_v5_sol_veg_obj_lod3_ImplicitRepresentation
_g_rh_v5_bdg_roofsurf_lod2_MultiSurface
_g_rh_v5_bdg_part_roofsurf_lod2_MultiSurface
_g_rh_v5_bdg_lod0_MultiSurface
_g_rh_v5_rel_tin_lod1_tin
rh_v4_bdg_lod2
Point Cloud
Animation
Loop
Camera Motion
_g_rh_v4_bdg_lod2
Orbit
Measure distance

Clicked coordinates
232405.18, 480565.57, 22.68

Layer
_g_rh_v4_bdg_lod2

Figure 3.2.: Use case 1 example - feature geometries viewed in 3D (Qgis2threejs plug-in)

3.2. Case 2: Users Interact with Visualised Feature Geometries and Retrieve the Attributes by Clicking on Features

Instead of merely visualising feature geometries, users may require the ability to interact with feature attributes by selecting individual geometries. Users can view or modify the corresponding attribute tables upon selection, depending on their assigned privileges. This spatial data interaction is similar to the Web Feature Service (WFS), where users can retrieve, modify, and exchange spatial data from remote databases using standardised operations over the Internet [34].

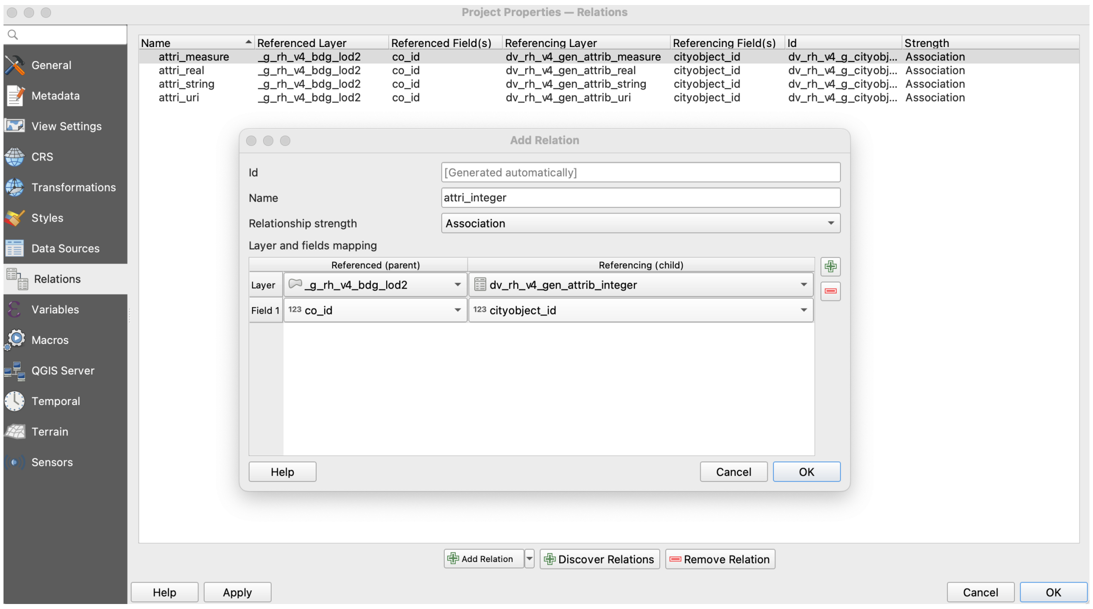
For CityGML data stored in 3DCityDB, accessing the feature attributes by clicking on geometries is possible by establishing relations in QGIS (Figure 3.3). Taking generic attributes as an example, these attributes can be associated with geometries, following use case 1, by adding relations in the QGIS project properties. Users can interact with feature generic attributes by selecting the corresponding feature geometry individually. Two sub-use cases arise based on the granted user privileges.

3.2.1. Access-only

In this sub-case, users are limited to viewing and inspecting the attributes of individual features. As shown in Figure 3.4, users can access the generic attributes by clicking on the geometries (with the selected feature highlighted in yellow), and the corresponding attributes of the feature are displayed in linked attribute tables.

3.2.2. Read and Write

In addition to the access-only sub-case, if the users are granted read and write privileges, the generic attributes will also become editable in the linked attribute tables (Figure 3.5). Changes made in QGIS will be updated back to the 3DCityDB, where the CityGML data is stored.



Name	Referenced Layer	Referenced Field(s)	Referencing Layer	Referencing Field(s)	Id	Strength
attri_measure	_g_rh_v4_bdg_lod2	co_id	dv_rh_v4_gen_attrib_measure	cityobject_id	dv_rh_v4_g_cityobj...	Association
attri_real	_g_rh_v4_bdg_lod2	co_id	dv_rh_v4_gen_attrib_real	cityobject_id	dv_rh_v4_g_cityobj...	Association
attri_string	_g_rh_v4_bdg_lod2	co_id	dv_rh_v4_gen_attrib_string	cityobject_id	dv_rh_v4_g_cityobj...	Association
attri_uri	_g_rh_v4_bdg_lod2	co_id	dv_rh_v4_gen_attrib_uri	cityobject_id	dv_rh_v4_g_cityobj...	Association

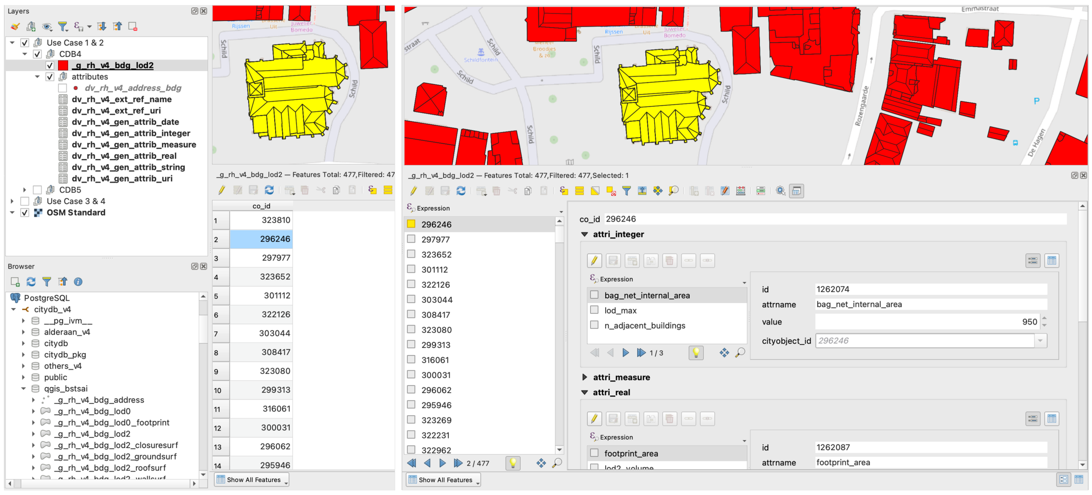
Add Relation

Id	[Generated automatically]
Name	attri_integer
Relationship strength	Association

Layer and fields mapping

	Referenced (parent)	Referencing (child)
Layer	_g_rh_v4_bdg_lod2	dv_rh_v4_gen_attrib_integer
Field 1	123 co_id	123 cityobject_id

Figure 3.3.: Use case 2 - Adding relations of feature attributes in QGIS



_g_rh_v4_bdg_lod2 — Features Total: 477,Filtered: 47

	co_id
1	323810
2	296246
3	297977
4	323652
5	301112
6	322126
7	303044
8	308417
9	323080
10	299313
11	316061
12	300031
13	296062
14	295946

_g_rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 1

296246
297977
323652
301112
322126
303044
308417
323080
299313
316061
300031
296062
295946
323269
322231
322962

co_id 296246

attri_integer

bag_net_internal_area
lod_max
n_adjacent_buildings

id	1262074
attrname	bag_net_internal_area
value	950
cityobject_id	296246

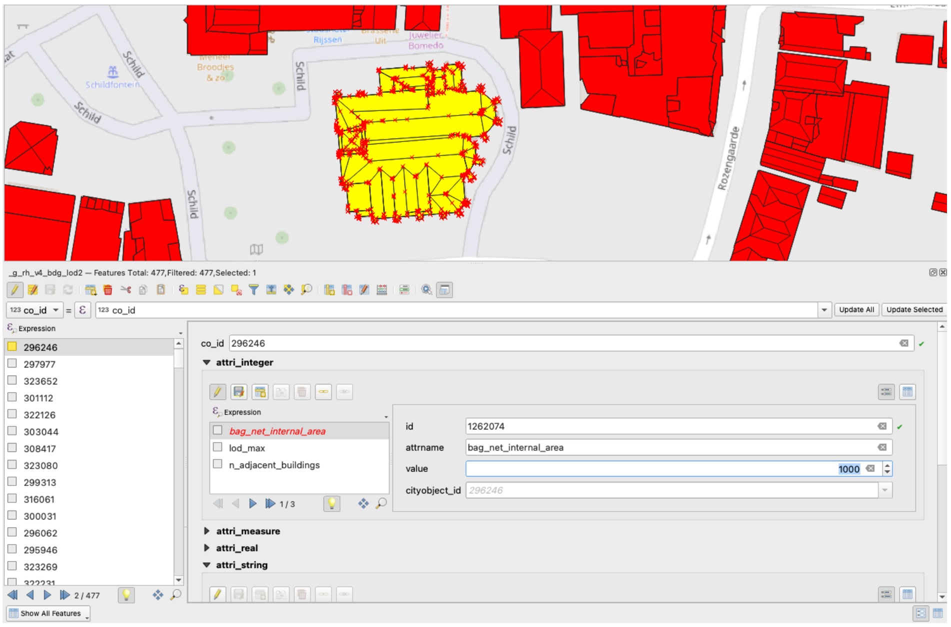
attri_measure

attri_real

footprint_area
lod2_volume

id	1262087
attrname	footprint_area

Figure 3.4.: Use case 2 example - attributes viewed by clicking on feature geometry (Left) attribute table view. (Right) attribute form view



_g_rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 1

co_id = co_id

296246
297977
323652
301112
322126
303044
308417
323080
299313
316061
300031
296062
295946
323269
322231

2 / 477

co_id: 296246

attri_integer

bag_net_internal_area
lod_max
n_adjacent_buildings

1 / 3

id	1262074
attrname	bag_net_internal_area
value	1000
cityobject_id	296246

attri_measure
attri_real
attri_string

Figure 3.5.: Use case 2 example - attributes editing in attribute form view

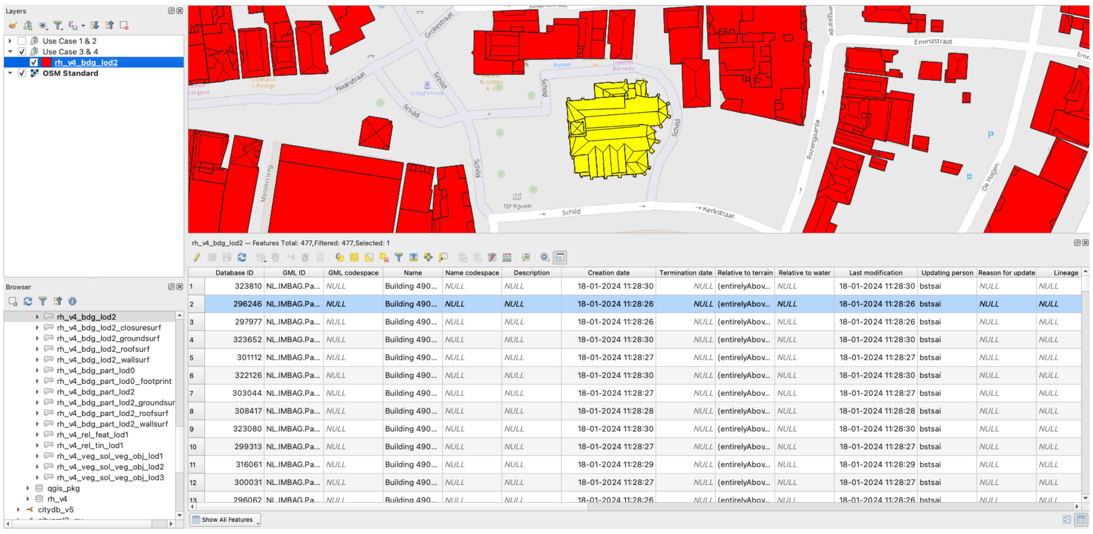
3.3. Case 3: Users Interact with Visualised Feature Geometries and the "Linked" Attributes

If the feature attributes are joined with different feature geometry representations, then users can have direct access to the feature attributes and can be processed as regular GIS layers (Figure 2.22). For CityGML data stored in 3DCityDB, the general and specific attributes can be directly accessed using the attribute table in QGIS as these attributes follow SFS model.

Joining the flattened attributes with feature geometries as GIS layers has several advantages, it enhances the attribute accessibility for user interaction, in which grouped selection based on certain query conditions is made possible using the select by expression function in the "attribute table view", users can edit and update attributes of selected featured at once. Moreover, feature layers following SFS model enable the support of QGIS built-in functions, allowing analyses such as spatial selection, geo-process operations and basic statistics for fields (columns), etc. Similar to use case 2, two sub-cases are derived based on the granted user privileges:

3.3.1. Access-only

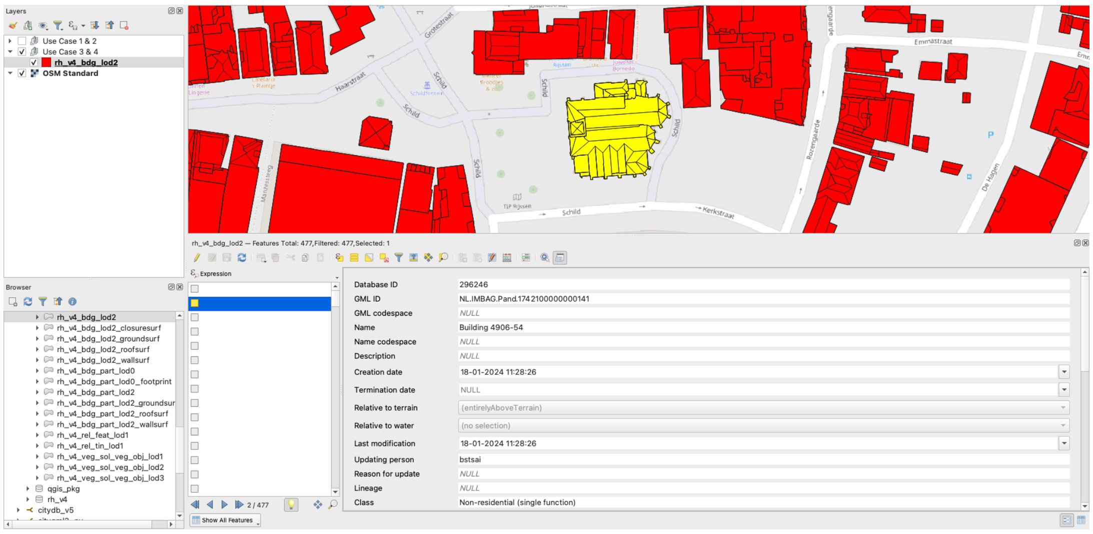
In the sub-case of use case 3, CityGML data encoded in 3DCityDB can be accessed as "normal" GIS layers that are created when importing data from shapefiles [35]. Users can access and inspect feature geometries and flattened (linearised) attributes, specifically the general and specific attributes, in both "attribute table view" (Figure 3.6) and "attribute form view" (Figure 3.7) in QGIS, enabling advanced queries supported by the expression selection and the usage of vector data processing tools.



rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 1

	Database ID	GML ID	GML codespace	Name	Name codespace	Description	Creation date	Termination date	Relative to terrain	Relative to water	Last modification	Updating person	Reason for update	Lineage
1	323810	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
2	296246	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:26	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:26	bstsai	NULL	NULL
3	297977	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:26	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:26	bstsai	NULL	NULL
4	323652	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
5	301112	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL
6	322126	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
7	303044	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL
8	308417	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:28	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:28	bstsai	NULL	NULL
9	323080	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
10	299313	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL
11	316061	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:29	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:29	bstsai	NULL	NULL
12	300031	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL
13	296062	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:26	NULL	(entirelyAbov...	NULL	18-01-2024 11:28:26	bstsai	NULL	NULL

Figure 3.6.: Use case 3 example - flattened (linearised) feature attributes viewed by clicking on feature geometry (attribute table view)



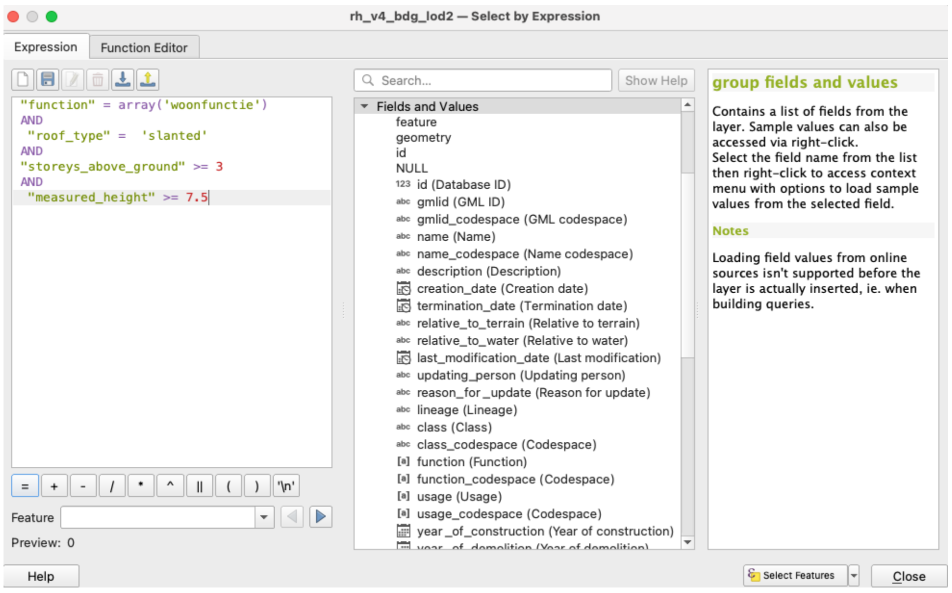
rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 1

Database ID	296246
GML ID	NL.IMBAG.Pand.1742100000000141
GML codespace	NULL
Name	Building 4906-54
Name codespace	NULL
Description	NULL
Creation date	18-01-2024 11:28:26
Termination date	NULL
Relative to terrain	(entirelyAboveTerrain)
Relative to water	(no selection)
Last modification	18-01-2024 11:28:26
Updating person	bstsai
Reason for update	NULL
Lineage	NULL
Class	Non-residential (single function)

Figure 3.7.: Use case 3 example - flattened (linearised) feature attributes viewed by clicking on feature geometry (attribute form view)

3.3.2. Read and Write

Batch editing of feature attributes is available if users are granted full read and write access in use case 3. For instance, users can perform conditional queries to select all buildings in LoD2 where the roof type is slanted, the building has at least 3 storeys above ground, the height is above 7.5 metres and the function is residential (Figure 3.8 and Figure 3.9). Then the selected buildings' attributes can be batch-edited using the field calculator (Figure 3.10). As shown in Figure 3.11, the ”function” values of the selected features are appended by the new value, ”newly added function”. GIS layers created from 3DCityDB in use case 3 facilitate the application of CityGML data, providing a straightforward platform for users to interact with 3D city models, regardless of the technical expertise for data querying and processing.



```
"function" = array('woonfunctie')
AND
 "roof_type" =  'slanted'
AND
"storeys_above_ground" >= 3
AND
 "measured_height" >= 7.5
```

Fields and Values
- feature
- geometry
- id
- NULL
- id (Database ID)
- gmlid (GML ID)
- gmlid_codespace (GML codespace)
- name (Name)
- name_codespace (Name codespace)
- description (Description)
- creation_date (Creation date)
- termination_date (Termination date)
- relative_to_terrain (Relative to terrain)
- relative_to_water (Relative to water)
- last_modification_date (Last modification)
- updating_person (Updating person)
- reason_for_update (Reason for update)
- lineage (Lineage)
- class (Class)
- class_codespace (Codespace)
- function (Function)
- function_codespace (Codespace)
- usage (Usage)
- usage_codespace (Codespace)
- year_of_construction (Year of construction)
- year_of_demolition (Year of demolition)

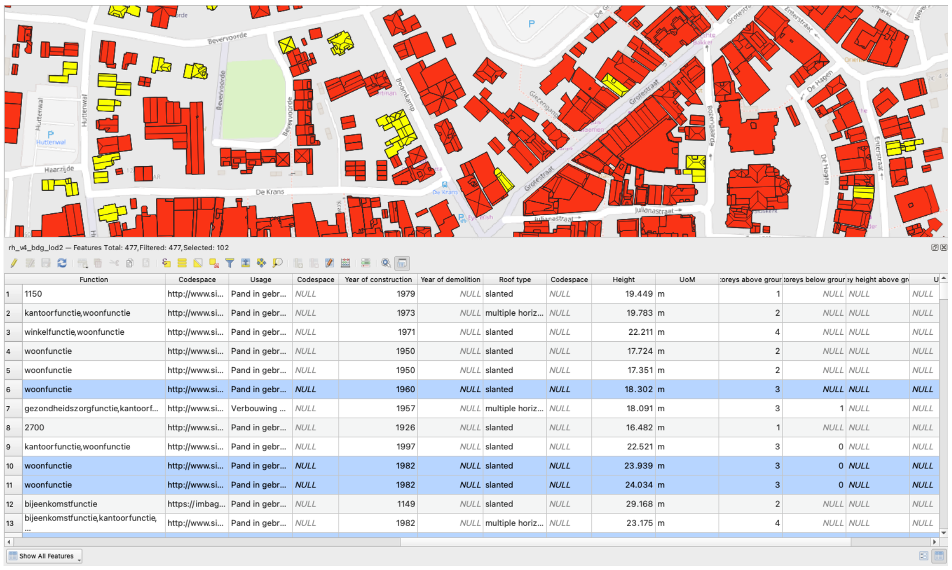
group fields and values

Contains a list of fields from the layer. Sample values can also be accessed via right-click. Select the field name from the list then right-click to access context menu with options to load sample values from the selected field.

Notes

Loading field values from online sources isn't supported before the layer is actually inserted, ie. when building queries.

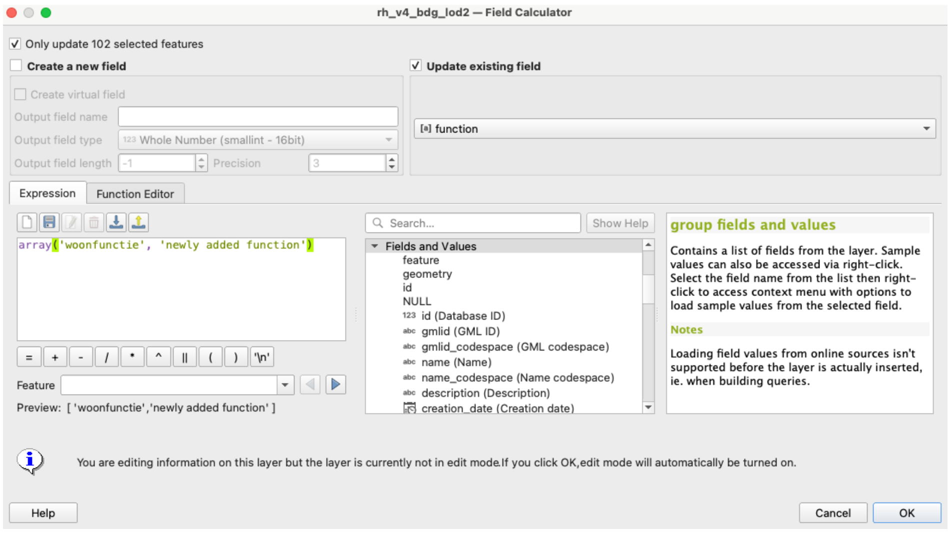
Figure 3.8.: Use case 3 example - conditional query



rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 102

	Function	Codespace	Usage	Codespace	Year of construction	Year of demolition	Roof type	Codespace	Height	UoM	toreys above grour	toreys below grour	ey height above gr	U
1	1150	http://www.si...	Pand in gebr...	NULL	1979	NULL	slanted	NULL	19.449	m	1	NULL	NULL	NULL
2	kantoorfunctie,woonfunctie	http://www.si...	Pand in gebr...	NULL	1973	NULL	multiple horiz...	NULL	19.783	m	2	NULL	NULL	NULL
3	winkelfunctie,woonfunctie	http://www.si...	Pand in gebr...	NULL	1971	NULL	slanted	NULL	22.211	m	4	NULL	NULL	NULL
4	woonfunctie	http://www.si...	Pand in gebr...	NULL	1950	NULL	slanted	NULL	17.724	m	2	NULL	NULL	NULL
5	woonfunctie	http://www.si...	Pand in gebr...	NULL	1950	NULL	slanted	NULL	17.351	m	2	NULL	NULL	NULL
6	woonfunctie	http://www.si...	Pand in gebr...	NULL	1960	NULL	slanted	NULL	18.302	m	3	NULL	NULL	NULL
7	gezondheidszorgfunctie,kantoorf...	http://www.si...	Verbouwing ...	NULL	1957	NULL	multiple horiz...	NULL	18.091	m	3	1	NULL	NULL
8	2700	http://www.si...	Pand in gebr...	NULL	1926	NULL	slanted	NULL	16.482	m	1	NULL	NULL	NULL
9	kantoorfunctie,woonfunctie	http://www.si...	Pand in gebr...	NULL	1997	NULL	slanted	NULL	22.521	m	3	0	NULL	NULL
10	woonfunctie	http://www.si...	Pand in gebr...	NULL	1982	NULL	slanted	NULL	23.939	m	3	0	NULL	NULL
11	woonfunctie	http://www.si...	Pand in gebr...	NULL	1982	NULL	slanted	NULL	24.034	m	3	0	NULL	NULL
12	bijeenkomstfunctie	https://imbag...	Pand in gebr...	NULL	1149	NULL	slanted	NULL	29.168	m	2	NULL	NULL	NULL
13	bijeenkomstfunctie,kantoorfunctie, ...	http://www.si...	Pand in gebr...	NULL	1982	NULL	multiple horiz...	NULL	23.175	m	4	NULL	NULL	NULL

Figure 3.9.: Use case 3 example - conditional query result (attribute table view)



rh_v4_bdg_lod2 — Field Calculator

- [✓] Only update 102 selected features
- [ ] Create a new field
- [✓] Update existing field

- [ ] Create virtual field

Output field name

Output field type: Whole Number (smallint - 16bit)

Output field length: -1 Precision: 3

function

Expression | Function Editor

```
array('woonfunctie', 'newly added function')
```

Feature

Preview: ['woonfunctie','newly added function']

Fields and Values
- feature
- geometry
- id
- NULL
- id (Database ID)
- gmlid (GML ID)
- gmlid_codespace (GML codespace)
- name (Name)
- name_codespace (Name codespace)
- description (Description)
- creation_date (Creation date)

group fields and values

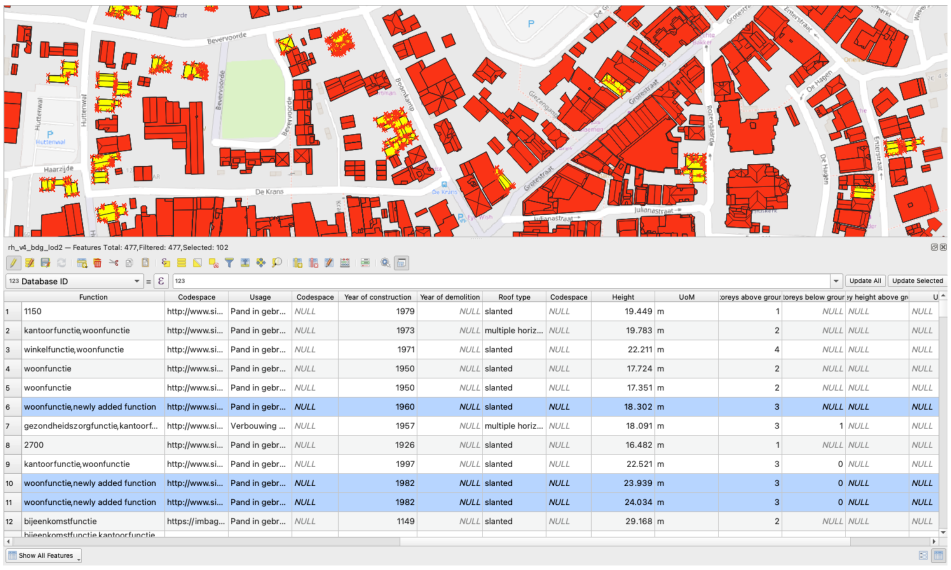
Contains a list of fields from the layer. Sample values can also be accessed via right-click. Select the field name from the list then right-click to access context menu with options to load sample values from the selected field.

Notes

Loading field values from online sources isn't supported before the layer is actually inserted, ie. when building queries.

You are editing information on this layer but the layer is currently not in edit mode.If you click OK,edit mode will automatically be turned on.

Figure 3.10.: Use case 3 example - field calculator for group-editing



rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 102

Database ID = 123

	Function	Codespace	Usage	Codespace	Year of construction	Year of demolition	Roof type	Codespace	Height	UoM	toreys above grour	toreys below grour	ey height above gr	U
1	1150	http://www.si...	Pand in gebr...	NULL	1979	NULL	slanted	NULL	19.449	m	1	NULL	NULL	NULL
2	kantoorfunctie,woonfunctie	http://www.si...	Pand in gebr...	NULL	1973	NULL	multiple horiz...	NULL	19.783	m	2	NULL	NULL	NULL
3	winkelfunctie,woonfunctie	http://www.si...	Pand in gebr...	NULL	1971	NULL	slanted	NULL	22.211	m	4	NULL	NULL	NULL
4	woonfunctie	http://www.si...	Pand in gebr...	NULL	1950	NULL	slanted	NULL	17.724	m	2	NULL	NULL	NULL
5	woonfunctie	http://www.si...	Pand in gebr...	NULL	1950	NULL	slanted	NULL	17.351	m	2	NULL	NULL	NULL
6	woonfunctie,newly added function	http://www.si...	Pand in gebr...	NULL	1960	NULL	slanted	NULL	18.302	m	3	NULL	NULL	NULL
7	gezondheidszorgfunctie,kantoorf...	http://www.si...	Verbouwing ...	NULL	1957	NULL	multiple horiz...	NULL	18.091	m	3	1	NULL	NULL
8	2700	http://www.si...	Pand in gebr...	NULL	1926	NULL	slanted	NULL	16.482	m	1	NULL	NULL	NULL
9	kantoorfunctie,woonfunctie	http://www.si...	Pand in gebr...	NULL	1997	NULL	slanted	NULL	22.521	m	3	0	NULL	NULL
10	woonfunctie,newly added function	http://www.si...	Pand in gebr...	NULL	1982	NULL	slanted	NULL	23.939	m	3	0	NULL	NULL
11	woonfunctie,newly added function	http://www.si...	Pand in gebr...	NULL	1982	NULL	slanted	NULL	24.034	m	3	0	NULL	NULL
12	bijeenkomstfunctie	https://imbag...	Pand in gebr...	NULL	1149	NULL	slanted	NULL	29.168	m	2	NULL	NULL	NULL
	bijeenkomstfunctie,kantoorfunctie													

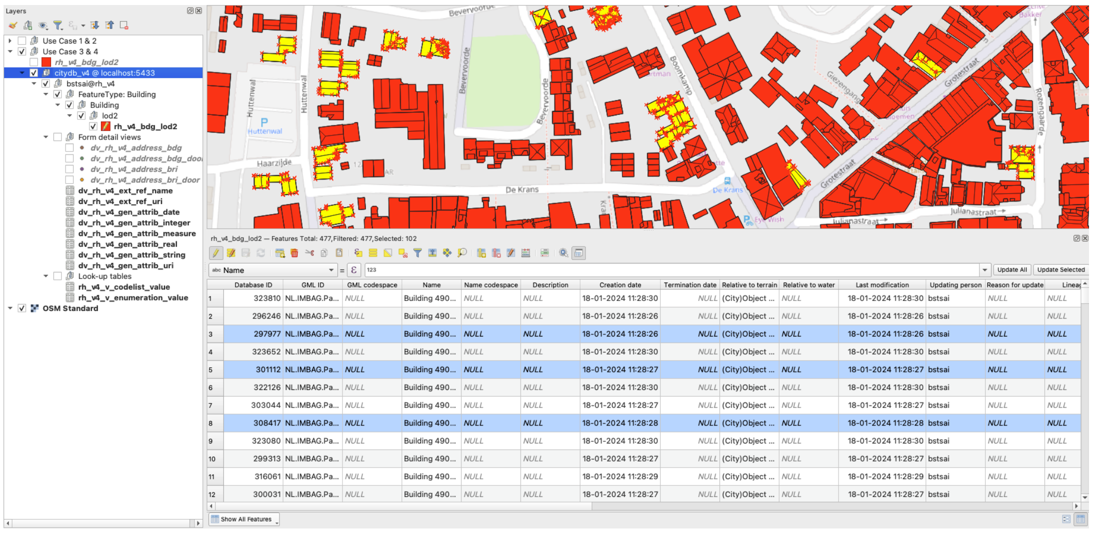
Figure 3.11.: Use case 3 example - group-edited result (attribute table view)

3.4. Case 4: Users Perform Use Case 3 Using GUIs in QGIS

The last use case is achieved by the front-end structure of the current plug-in, and its introduction can be seen in Section 2.4. Users can connect to the 3DCityDB v.4.x instances via “QGIS Package Administrator” (see Figure 2.23) and be granted full read and write access. After setting up the database connection, the GIS layers can be imported using ”Layer Loader” (see Figure 2.24), and users can select the desired feature and LoD representations. The selected

GIS layer is imported under a hierarchical table of content together with related generic attribute views (Figure 3.12) for detail form view and lookup tables for checking the codelist and enumeration values.

With the loaded GIS feature layers, users can interact with the CityGML data stored in 3DCityDB more intuitively. Batch-edit is enabled via ”attribute table view” while details of the generic attributes can be inspected and edited when users click on individual feature geometry in ”attribute form view” (see Figure 2.27).



rh_v4_bdg_lod2 — Features Total: 477,Filtered: 477,Selected: 102

	Database ID	GML ID	GML codespace	Name	Name codespace	Description	Creation date	Termination date	Relative to terrain	Relative to water	Last modification	Updating person	Reason for update	Lineag
1	323810	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(City)Object ...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
2	296246	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:26	NULL	(City)Object ...	NULL	18-01-2024 11:28:26	bstsai	NULL	NULL
3	297977	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:26	NULL	(City)Object ...	NULL	18-01-2024 11:28:26	bstsai	NULL	NULL
4	323652	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(City)Object ...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
5	301112	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(City)Object ...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL
6	322126	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(City)Object ...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
7	303044	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(City)Object ...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL
8	308417	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:28	NULL	(City)Object ...	NULL	18-01-2024 11:28:28	bstsai	NULL	NULL
9	323080	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:30	NULL	(City)Object ...	NULL	18-01-2024 11:28:30	bstsai	NULL	NULL
10	299313	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(City)Object ...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL
11	316061	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:29	NULL	(City)Object ...	NULL	18-01-2024 11:28:29	bstsai	NULL	NULL
12	300031	NL.IMBAG.Pa...	NULL	Building 490...	NULL	NULL	18-01-2024 11:28:27	NULL	(City)Object ...	NULL	18-01-2024 11:28:27	bstsai	NULL	NULL

Figure 3.12.: Use case 4 example - conditional query result (attribute table view)

3.5. Challenges for Possible QGIS Use Cases Using CityGML Data in the 3DCityDB v.5.0

From the analysis of the use cases, it results that there are two main challenges to be solved to use the spatial data stored within the 3DCityDB v.5.0:

- **Feature geometry extraction**
  This part involves extracting all possible combinations of feature geometry representations from the CityGML data encoded in 3DCityDB v.5.0, which will be the basis for joining all corresponding attributes.
- **Feature attribute flattening**
  This part focuses on flattening the attributes of CityGML data encoded in the PROPERTY tables of 3DCityDB v.5.0. This will be the main challenge of this research since all attribute types (general, specific and generic) need to be flattened, unlike the encoding of 3DCityDB v.4.x, where the general and specific attributes are already linearised.

4. Methodology

This chapter presents the details of the methodology applied in this research regarding the reproduction of GIS layers creation from CityGML data stored in 3DCityDB v.5.0. As described in Section 2.4, the current plug-in server-side functions join feature geometry in materialized views with feature attributes, these two main parts are stored as views for GIS layer creation. A similar approach will be applied to generate GIS layers from the 3D models encoded in3DCityDB schemas v.5.0.

The overall methodology of this research is structured into four main parts as shown in Figure 4.1:

- **Preparation**: This initial phase involves installing the QGIS Package (qgis_pkg for short), which is a set of PL/pgSQL functions developed by this research to perform relevant SQL queries for GIS layers creation from 3DCityDB v.5.0.
- **Feature geometry**: It consists of two steps: checking the existence of the available feature geometry representations within the target schema and updating the feature geometry metadata table. Users can view the metadata table to create views or materialized views of their desired feature geometry representations.
- **Feature attribute**: Similar to the feature geometry part, this involves checking the existence of available feature attributes within the target schema. The information on the found attributes will be updated in the feature attribute metadata table. Users can select their desired feature attributes based on the feature geometry and store them as views or materialized views.
- **GIS layers creation**: With the views of feature geometries and attributes, users can then start to generate GIS layers by joining them. This involves different joining approaches, which will be detailed later in this research.

Section 4.1 introduces the preparation need to meet requirements for GIS layers generation and explains the necessity of the feature geometry and attribute metadata tables regarding the 3DCityDB v.5.0 encoding. Section 4.2 describes the working principle to scan and retrieve all existing geometry representations and LoDs of the features in the target 3DCityDB v.5.0 schema, followed by the query time experiment for the decision of geometry view generation. Section 4.3 elaborates on the attribute types observed from the mapping of feature attributes in 3DCityDB v.5.0 and then introduces the methods that are used to collect and flatten (linearise) the attributes to form the views that will be joined with the feature geometries. Finally, Section 4.4 discusses different approaches to join the views of feature geometries and attributes, including a query time experiment to justify the choice for creating GIS layers.

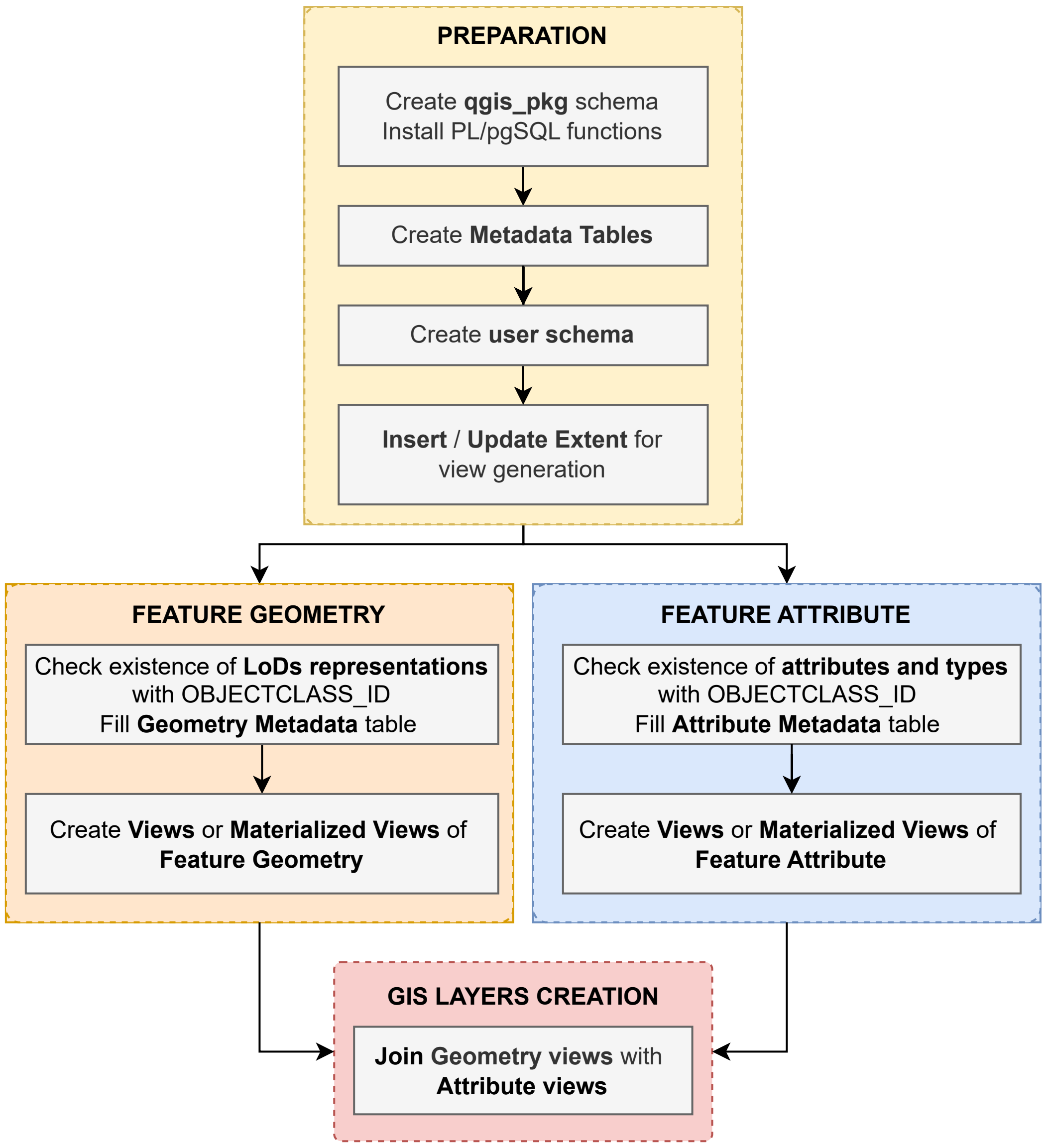
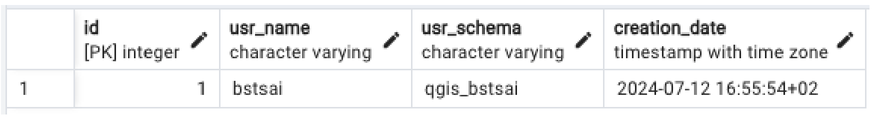

PREPARATION
Create qgis_pkg schema
Install PL/pgSQL functions
Create Metadata Tables
Create user schema
Insert / Update Extent for view generation
FEATURE GEOMETRY
Check existence of LoDs representations with OBJECTCLASS_ID
Fill Geometry Metadata table
Create Views or Materialized Views of Feature Geometry
FEATURE ATTRIBUTE
Check existence of attributes and types with OBJECTCLASS_ID
Fill Attribute Metadata table
Create Views or Materialized Views of Feature Attribute
GIS LAYERS CREATION
Join Geometry views with Attribute views

Figure 4.1.: Overview of the methodology

4.1. Preparation

The PL/pgSQL functions used for GIS layer creation in this research are included in the QGIS package. Users are required to install it on top of 3DCityDB v.5.0 instances for data retrieval. After a successful package installation, a new schema named `qgis_pkg` will be created under the connected PostgreSQL database, which contains the following tables:

- **USR_SCHEMA**: It stores the metadata of the user-defined schemas (Figure 4.2).



	id [PK] integer	usr_name character varying	usr_schema character varying	creation_date timestamp with time zone
1	1	bstsai	qgis_bstsai	2024-07-12 16:55:54+02

Figure 4.2.: Example of USR_SCHEMA table created in qgis_pkg

- **ATTRIBUTE_DATATYPE_LOOKUP**: The encoding rules for feature attributes are detailed in the `schema` column of the DATATYPE table in 3DCityDB v.5.0. This column provides information about the data types of feature attributes and how their values are stored in the PROPERTY table. By examining the DATATYPE table, it can be seen that attributes are stored as simple or complex attributes.

  Listing 4.1 illustrates the explicit storage of the "Integer" attribute data type. The `identifier` key specifies the attribute data type and the CityGML modules where this data type is defined. The `table` key gives the table name storing the attribute, while the `value` key details the column where the attribute values can be found.

  Listing 4.2 illustrates the implicit storage of the "Height" attribute data type defined within the "Construction" module in CityGML v.3.0. Unlike the "Integer" type, the values of "Height" are represented by multiple related child attributes stored explicitly. The `properties` key specifies the mapping details of all the related child attributes, including their types and the association with their parent attribute. This association is achieved by joining the `parent_ids` of the child attributes with the `id` of the parent attribute. Feature attributes following the explicit and implicit mapping rules are categorised as "Inline" and "Nested" attributes, respectively, within this research. Section 4.3.1.1 provides detailed elaboration on the characteristics of "Inline" and "Nested" attributes.

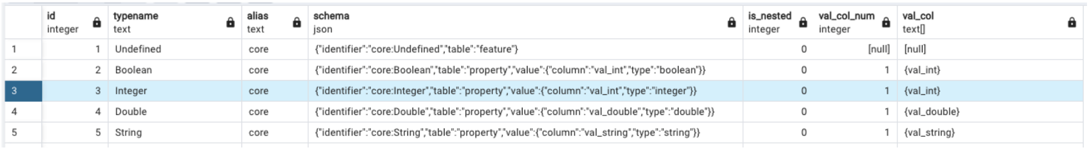
  The ATTRIBUTE_DATATYPE_LOOKUP table is derived from the DATATYPE table in 3DCityDB v.5.0, serving as an advanced reference as depicted in Figure 4.3. It provides essential information for the PL/pgSQL functions to distinguish attribute classes and their corresponding value columns.

```
{
     "identifier" : "core:Integer",
     "table" : "property",
     "value":  {
         "column": "val_int",
         "type": "integer"
     }
}
```

Listing 4.1: Mapping example in JSON - Attribute DataType: "Integer"

```json
{
    "identifier" : "con:Height",
    "table" : "property",
    "value":  {
        "property":  0
    },
    "properties" : [
        {
            "name":  "value",
            "namespace" :  "http://3dcitydb.org/3dcitydb/construction/5.0",
            "type":  "core:Measure",
            "join":  {
                "table" :  "property",
                "fromColumn" :  "parent_id",
                "toColumn" :  "id"
            }
        },
        ...
        {
            "name":  "highReference",
            "namespace" :  "http://3dcitydb.org/3dcitydb/construction/5.0",
            "type":  "core:Code",
            "join":  {
                "table" :  "property",
                "fromColumn" :  "parent_id",
                "toColumn" :  "id"
            }
        }
    ]
}
```

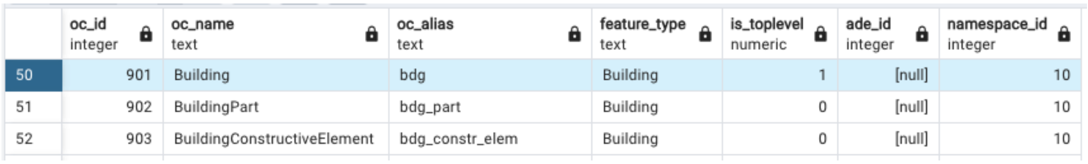
Listing 4.2: Mapping example in JSON - Attribute DataType: "Height"



	id integer	typename text	alias text	schema json	is_nested integer	val_col_num integer	val_col text[]
1	1	Undefined	core	{"identifier":"core:Undefined","table":"feature"}	0	[null]	[null]
2	2	Boolean	core	{"identifier":"core:Boolean","table":"property","value":{"column":"val_int","type":"boolean"}}	0	1	{val_int}
3	3	Integer	core	{"identifier":"core:Integer","table":"property","value":{"column":"val_int","type":"integer"}}	0	1	{val_int}
4	4	Double	core	{"identifier":"core:Double","table":"property","value":{"column":"val_double","type":"double"}}	0	1	{val_double}
5	5	String	core	{"identifier":"core:String","table":"property","value":{"column":"val_string","type":"string"}}	0	1	{val_string}

Figure 4.3.: Example of the ATTRIBUTE_DATATYPE_LOOKUP table created in qgis_pkg

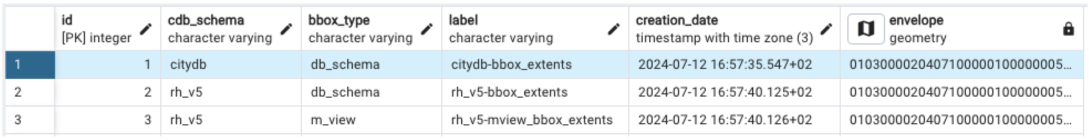
- **CLASSNAME_LOOKUP**: This is the lookup table derived from the OBJECTCLASS table in 3DCityDB v.5.0 (Figure 4.4). It stores the `objectclass_id`, the corresponding class names and the class name aliases defined by this research. The class name aliases are referenced to name the feature geometry views.



	oc_id integer	oc_name text	oc_alias text	feature_type text	is_toplevel numeric	ade_id integer	namespace_id integer
50	901	Building	bdg	Building	1	[null]	10
51	902	BuildingPart	bdg_part	Building	0	[null]	10
52	903	BuildingConstructiveElement	bdg_constr_elem	Building	0	[null]	10

Figure 4.4.: Example of the CLASSNAME_LOOKUP table created in qgis_pkg

- **FEATURE_GEOMETRY_METADATA_TEMPLATE**: This is the template table that will be duplicated to the newly created user schemas. The duplicated table is created under the user schemas for the users to check and store the metadata of the existing feature LoDs and geometry representations, which is elaborated in Section 5.3.
- **FEATURE_ATTRIBUTE_METADATA_TEMPLATE**: This is the template table that will be duplicated to the newly created user schemas. The duplicated table is created under the user schemas for the users to check and store the metadata of existing feature attributes, which is elaborated in Section 5.4.
- **EXTENTS_TEMPLATE**: This is the template table that will be duplicated to the newly created user schemas for storing the metadata and geometries of the bounding boxes for operations. Users can insert or update the bounding boxes to limit the extent of GIS layers creation (Figure 4.5).



	id [PK] integer	cdb_schema character varying	bbox_type character varying	label character varying	creation_date timestamp with time zone (3)	envelope geometry
1	1	citydb	db_schema	citydb-bbox_extents	2024-07-12 16:57:35.547+02	01030000204071000001000000005...
2	2	rh_v5	db_schema	rh_v5-bbox_extents	2024-07-12 16:57:40.125+02	01030000204071000001000000005...
3	3	rh_v5	m_view	rh_v5-mview_bbox_extents	2024-07-12 16:57:40.126+02	01030000204071000001000000005...

Figure 4.5.: Example of the EXTENTS table created in user-created schema

- **LAYER_METADATA_TEMPLATE**: This is the template table which will be duplicated to the newly created user schemas. The duplicated table is created under the user schemas for storing the metadata of the created GIS layers, allowing users to manage the GIS layers, which is elaborated in Section 6.3.

The reasons for creating metadata tables of feature geometry and attribute correspond to the changes in the mapping rule of 3DCityDB v.5.0. In 3DCityDB v.4.x, feature geometry representations are encoded in different thematic class tables (see Section 2.2.1). The current plug-in traverses these classes across all thematic modules to collect possible feature geometry representations as materialized views. However, in 3DCityDB v.5.0, the feature registry and spatial properties are integrated into the FEATURE and PROPERTY tables (see Section 2.2.2). Traversing existing classes to collect all available feature geometry representations by cross-referencing these two tables could be time-consuming. Even with limited combinations of representation, querying from a massive dataset could still take a long time.

To enhance the query time efficiency given the characteristics of the 3DCityDB v.5.0 encoding, creating metadata tables to store existing feature geometry representations and attributes is proposed. The general concept is to perform a schema-wise scan of the target dataset to obtain the distinct combinations of feature classes and their corresponding geometry representations, along with attributes. These distinct pairs of records will then be updated to the feature geometry and attribute metadata tables, serving as menus for users to create GIS layers. Users can specify the desired feature geometry representation to be joined with the selected attributes to form the GIS layer and import it into QGIS for interaction. This approach avoids the repetitive joining of FEATURE and PROPERTY tables, which improves the query time efficiency and enhances flexibility for GIS layer creation.

With the `qgis_pkg` installed and the relevant tables set up, users can create the user schema to store the extents and the views generated by the PL/pgSQL functions. The name of the user schema is determined by the user's name prefixed with "qgis_". For example, "qgis_postgres" is the default name of the newly created user schema, and ”postgres” is the default user name in PostgreSQL.

After creating user schemas, users are required to specify the extent. By default, the full dataset extent is computed and applied in further operations; however, users can specify the minimum and maximum xy-coordinates of the bounding box based on the Spatial Reference Identifier (SRID) of the target database. The specified extents will be stored in the EXTENTS table, duplicated from the EXTENTS_TEMPLATE table in the qgis_pkg when the user schema is created. The extent selection improves the query performance of metadata checks and GIS layer creation, reducing the time needed for feature spatial properties check, generating views and refreshing materialized views in case of large datasets.

4.2. Feature Geometry

4.2.1. Check Existence of Feature Geometry Representations and LoDs

The first step in the feature geometry process is to check the existence of feature geometry representations and LoDs within the user-selected extent. Users can perform a schema-wise scan of the dataset or specify the bounding box for a smaller, faster check. Since features in CityGML v.3.0 inherit the geometry and LoD concept from the Core module (see Section 2.1.2), possible feature geometry representations and LoDs only vary depending on the type of feature, such as space features, boundary features or relief features.

Table 4.1 shows the geometry and LoD concept of the “AbstractSpace” and the “AbstractThematicSurface” classes in CityGML v.3.0 Core module (see Figure 2.6), which defines all the possible geometry representations and LoDs of space and boundary features.

For the features in the “Relief” module, since they are derived from a higher level abstract class, the “AbstractSpaceBoundary” class, they have their own spatial properties specifications (see Figure 2.7). These include:

- **Relief Feature**: This is derived from the ”ReliefFeature” class, which is the bounding box of the relief component features. Its geometry is represented by the envelope stored in the FEATURE table in 3DCityDB v.5.0 with LoD1-3.
- **Relief Component Feature**: This is derived from the ”AbstractRelifComponent” class, which is the abstract class for four different types of relief component features with LoD0-3. The spatial properties of the relief component features are:
    - **TINRelief**: ”tin”.
    - **MassPointRelief**: “reliefPoint”.
    - **BreaklineRelief**: ”ridgeOrValleyLines” and ”breaklines”
    - **RasterRelief**: ”grid”

Notice that the “RasterRelief” class is excluded by the current plug-in as it is not supported by 3DCityDB v.4.x. It is also excluded from this research as raster-based features from the 3DCityDB v.5.0 are not supported.

Class	LoD	Geometry Representation	Class	LoD	Geometry Representation
Abstract Space (space feature)	0	Point	**Abstract Thematic Surface** (boundary feature)	0	MultiSurface
	0	MultiSurface		0	MultiCurve
	0	MultiCurve		1	MultiSurface
	1	Solid		2	MultiSurface
	1	TerrainIntersectionCurve		3	MultiSurface
	1	ImplicitRepresentation		x	pointCloud
	2	Solid		x	envelope
	2	MultiSurface			
	2	MultiCurve			
	2	TerrainIntersectionCurve			
	2	ImplicitRepresentation			
	3	Solid			
	3	MultiSurface			
	3	MultiCurve			
	3	TerrainIntersectionCurve			
	3	ImplicitRepresentation			
	x	pointCloud			
	x	envelope			

[1] The "x" in the table indicates no specific LoDs of that geometry type.
[2] The "envelope" is included in the table for the 'ReliefFeature" class.

Table 4.1.: Possible geometry representations and LoDs of space and boundary features

After identifying features' spatial properties in 3DCityDB v.5.0, it can be concluded that 4 feature types determine the possible geometry representations and LoDs: space, boundary, relief and relief component features. To store the distinct combinations of feature geometry representations and LoDs from the target 3DCityDB v.5.0 schema, the feature geometry metadata table should include at least the following columns:

1. **Schema Name**: Identifier of the target 3DCityDB schemas.
2. **Parent Objectclass_id**: Identifier of the parent space feature of boundary features, eg., the roofs of buildings. For space features, this should be empty or null.
3. **Objectclass_id**: Identifier of the feature classes.
4. **Geometry Name**: Represents the spatial properties of features in 3DCityDB v.5.0 PROPERTY table, storing information about the geometry representation and its LoD from Table 4.1, except relief features. Example value entries could be "lod1Solid" or "lod2MultiSurface".

With the extent specifications, users can perform either a complete scan or an optional extent selection scan on the target dataset. The feature geometry metatable will be populated with distinct feature geometry representations and LoDs records. This metadata table serves as a reference menu for users to create geometry views.

4.2.2. Create Views or Materialized Views for Feature Geometry

With the FEATURE_GEOMETRY_METADATA table checked and updated, users can retrieve feature geometries by specifying their desired schema, parent feature class (for boundary features), feature class, and feature LoDs geometry representations. Feature geometries are collected using the specified value pairs to cross-reference the relevant tables. An example of this process is shown in Listing 2.2. The collected feature geometries can be stored as views or materialized views.

According to Section 2.4, materialized views are chosen for the feature geometries due to the complexity involved in decomposing and storing feature geometries in 3DCityDB v.4.x. The choice aims to improve user experience, although it consumes storage space and requires time to generate and refresh them when creating GIS layers [15]. However, in 3DCityDB v.5.0, feature geometries are no longer decomposed; they are stored directly in GEOMETRY_DATA table without decomposition and are associated with spatial properties via geometry or implicit geometry property keys in the PROPERTY table (see Section 2.2.2). Consequently, a query time assessment is performed to check the query time from the datasets encoded within different versions of 3DCityDB to determine the approach for storing feature geometries extracted from the data stored within 3DCityDB v.5.0.

Table 4.2 and Table 4.3 present the query time comparison for a full dataset extent selection on feature geometry views and materialized views from the "Rijsen-Holten" and "Vienna" datasets, encoded within 3DCityDB versions v.4.x and 5.0. Since the pre-aggregation for querying feature geometries is eliminated in 3DCityDB v.5.0, which theoretically improves the query speed. However, the integration of FEATURE and PROPERTY tables in 3DCityDB v.5.0 can result in massive table sizes, potentially undermining the anticipated query speed improvements. Analysing the query times for different feature types reveals specific considerations:

- **Space Feature**: The query time difference between view types is minimal. However, for classes such as 'SolitaryVegetationObject' (tree) with implicit representations, using materialized views is recommended due to the longer time required to query implicit reference information from a large PROPERTY table.
- **Boundary Feature**: As the boundary feature relations are stored in the PROPERTY table, querying the Space-Boundary feature hierarchy involves repetitive cross-referencing between FEATURE and PROPERTY tables. Materialized views offer significantly faster query times in this scenario.
- **Relief Feature**: Views suffice for querying a small number of relief extents efficiently.
- **Relief Component Feature**: Materialized views are not essential here, as the query times between view types are comparable.

In conclusion, while 3DCityDB v.5.0 eliminates the need for pre-aggregation when querying feature geometries, materialized views remain preferable. Although materialized views consume memory storage space and may take longer to create and refresh, they enhance the user query experience by avoiding extensive cross-referencing of large tables, thereby ensuring quick access to desired results.

Feature Geometry Representations and LoDs					3DCityDB v.4.x			3DCityDB v.5.0			Number of feature selected
					View	Materialized View		View	Materialized View		
Dataset	Class (objectclass_id)		LoD	Geometry Representation	Query time	Query time	Refresh time	Query time	Query time	Refresh time	
Rijsen-Holten	Building (901)		0	Multi Surface	00:01.0	00:00.2	00:03.1	00:02.7	00:01.8	00:04.1	30,449
	BuildingPart (902)		0	Multi Surface	00:00.1	00:00.1	00:00.1	00:00.1	00:00.0	00:00.1	107
	Building (901)	ClosureSurface (15)	2	Multi Surface	00:01.1	00:00.2	00:02.9	00:03.5	00:00.3	00:05.8	50,835
		GroundSurface (710)	2	Multi Surface	00:01.0	00:00.2	00:02.6	00:03.5	00:00.3	00:08.2	29,511
		RoofSurface (712)	2	Multi Surface	00:01.5	00:00.4	00:04.5	00:03.5	00:00.7	00:06.5	93,799
		WallSurface (709)	2	Multi Surface	00:05.3	00:01.1	00:16.2	00:04.4	00:03.6	01:13.2	555,459
	BuildingPart (902)	ClosureSurface (15)	2	Multi Surface	00:01.1	00:00.2	00:02.9	00:00.2	00:00.0	00:00.2	67
		GroundSurface (710)	2	Multi Surface	00:01.0	00:00.2	00:02.6	00:00.1	00:00.0	00:00.2	107
		RoofSurface (712)	2	Multi Surface	00:01.5	00:00.4	00:04.5	00:00.1	00:00.0	00:00.2	411
		WallSurface (709)	2	Multi Surface	00:05.3	00:01.1	00:16.2	00:00.3	00:00.0	00:00.6	2,903
	SolitaryVegetationObject (1301)		1	Implicit Representation	00:04.0	00:00.8	00:09.3	03:21.4	00:02.8	06:05.1	58,515
			2	Implicit Representation	00:03.8	00:00.6	00:06.3	03:16.5	00:02.5	06:04.8	58,515
			3	Implicit Representation	00:40.9	00:07.2	01:24.7	03:47.1	00:09.1	06:44.8	58,515
	ReliefFeature (500)		1	Envelope	00:00.1	00:00.1	00:00.1	00:00.0	00:00.0	00:00.0	1
	TINRelief (502)		1	tin	00:06.7	00:02.7	00:15.2	00:05.8	00:05.1	00:09.4	4,941

time unit: mm:ss.s

Table 4.2.: Comparison of the query time from geometry views and materialized views in 3DCityDB v.4.x and v.5.0 (Rijsen-Holten dataset)

Feature Geometry Representations and LoDs					3DCityDB v.4.x			3DCityDB v.5.0			Number of feature selected
					View	Materialized View		View	Materialized View		
Dataset	Class (objectclass_id)		LoD	Geometry Representation	Query time	Query time	Refresh time	Query time	Query time	Refresh time	
Vienna	Building (901)	GroundSurface (710)	2	Multi Surface	00:27.5	00:01.6	01:12.1	00:46.5	00:00.6	01:37.5	95,018
		RoofSurface (712)	2	Multi Surface	00:36.5	00:02.6	01:27.0	00:56.3	00:01.5	02:04.6	300,593
		WallSurface (709)	2	Multi Surface	00:39.3	00:09.8	02:11.0	01:45.7	00:06.8	02:17.3	961,252
	BuildingPart (902)	GroundSurface (710)	2	Multi Surface	00:27.5	00:01.6	01:12.1	00:58.7	00:03.3	02:09.9	382,225
		RoofSurface (712)	2	Multi Surface	00:36.5	00:02.6	01:27.0	01:06.7	00:07.2	02:18.1	882,590
		WallSurface (709)	2	Multi Surface	00:39.3	00:09.8	02:11.0	01:16.9	00:12.4	03:06.9	4,085,091
	SolitaryVegetationObject (1301)		2	Implicit Representation	00:08.9	00:01.4	00:17.2	00:08.2	00:01.4	00:16.4	187,360
	ReliefFeature (500)		2	envelope	00:00.1	00:00.2	00:00.2	00:00.1	00:00.1	00:00.1	1,815
			3	envelope	00:00.1	00:00.4	00:00.1	00:00.1	00:00.2	00:00.1	1,815
	TINRelief (502)		2	tin	01:06.8	00:06.0	02:10.8	00:14.0	00:13.3	00:23.1	1,815
			3	tin	01:28.4	00:15.3	03:01.1	00:33.6	00:32.9	00:48.0	1,815
			4	tin	02:01.4	00:28.0	04:16.8	01:03.4	01:00.3	01:26.1	1,815

time unit: mm:ss.s

Table 4.3.: Comparison of the query time from geometry views and materialized views in 3DCityDB v.4.x and v.5.0 (Vienna dataset)

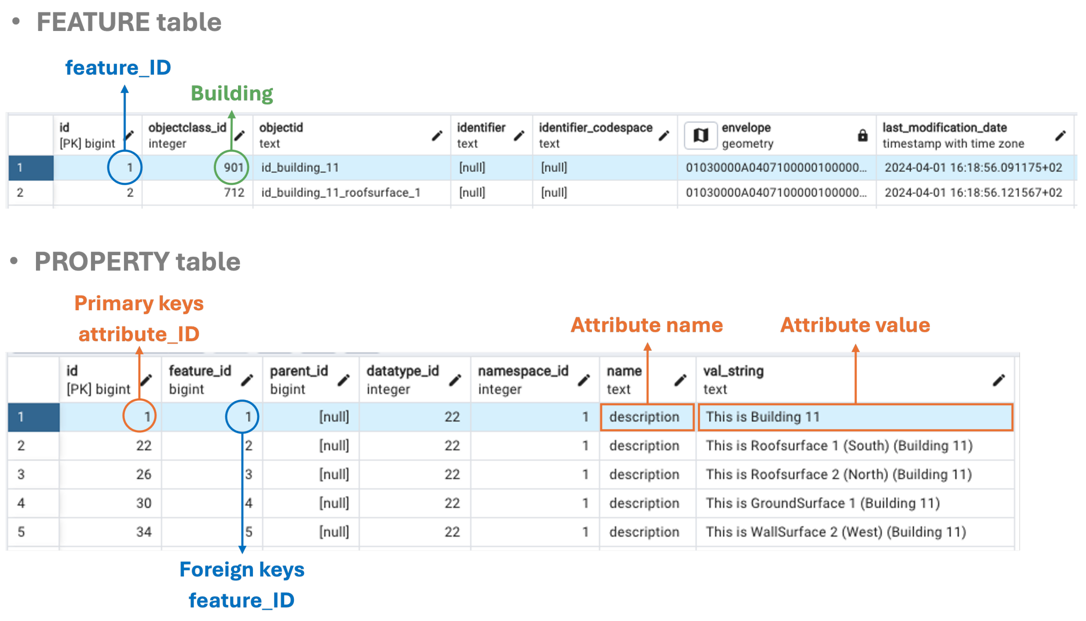
4.3. Feature Attributes

4.3.1. Check Existence of Feature Attributes

The first step in the feature attribute process is to check the existence of feature attributes within the user-selected extent. Unlike 3DCityDB v.4.x, where general, specific and generic attributes are stored in different tables (see Section 2.2.1), feature attributes in 3DCityDB v.5.0 are all stored in the PROPERTY table following the EAV model. It reduces the complexity of the 3DCityDB by eliminating the need for thematic abstract class tables such as BUILDING, thus simplifying table cross-referencing. However, this also results in the substantially larger PROPERTY table, which hinders the query performance and results in attribute flattening (or linearisation) for creating GIS layers. The following subsections outline the key factors to query and flatten feature attributes regarding the encoding of 3DCityDB v.5.0.

4.3.1.1. "Inline" and "Nested" Attributes

Feature attributes are integrated into the PROPERTY table in 3DCityDB v.5.0, where each attribute is typically represented as a single row. Each row in the PROPERTY table records an attribute related to a feature, identified by the `feature_id`. The `datatype_id` indicates the attribute data type, and the attribute value is stored in the corresponding value column depending on the attribute data type. Attributes stored in this format are termed as the "Inline" attributes in this research, which can be illustrated with an example such as the "description" of buildings in Figure 4.6. In this example, each row in the table represents a building "description", which is a string-type attribute, with its value stored in the `val_string` column.



- FEATURE table

	id [PK] bigint	objectclass_id integer	objectid text	identifier text	identifier_codespace text	envelope geometry	last_modification_date timestamp with time zone
1	1	901	id_building_11	[null]	[null]	01030000A0407100000100000...	2024-04-01 16:18:56.091175+02
2	2	712	id_building_11_roofsurface_1	[null]	[null]	01030000A0407100000100000...	2024-04-01 16:18:56.121567+02

- PROPERTY table

	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	namespace_id integer	name text	val_string text
1	1	1	[null]	22	1	description	This is Building 11
2	22	2	[null]	22	1	description	This is Roofsurface 1 (South) (Building 11)
3	26	3	[null]	22	1	description	This is Roofsurface 2 (North) (Building 11)
4	30	4	[null]	22	1	description	This is GroundSurface 1 (Building 11)
5	34	5	[null]	22	1	description	This is WallSurface 2 (West) (Building 11)

Figure 4.6.: Inline attribute example - "description" of buildings

Apart from the inline attribute, 3DCityDB v.5.0 applies attribute storage rules that align with the complex data types introduced by CityGML v.3.0. For example, the "Height" data type within the "Construction" module specifies properties like the "height" of buildings, bridges and tunnels, which includes 4 child attributes: "heightReference", "lowReference", "status" and "value" (Figure 4.7). In 3DCityDB v.5.0 PROPERTY table, a single entry for a building's height consists of a parent row followed by multiple related child attributes, linked by `parent_id` as shown in Figure 4.8. The child attribute values are stored across different value columns based on their attribute types and are associated with a specific feature through the `feature_id`. This format of storing feature attributes is referred to as the "Nested" attributes in this research.

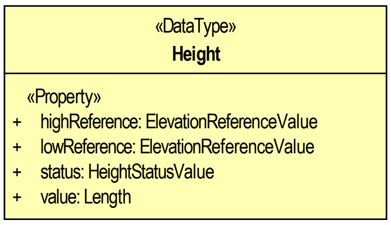
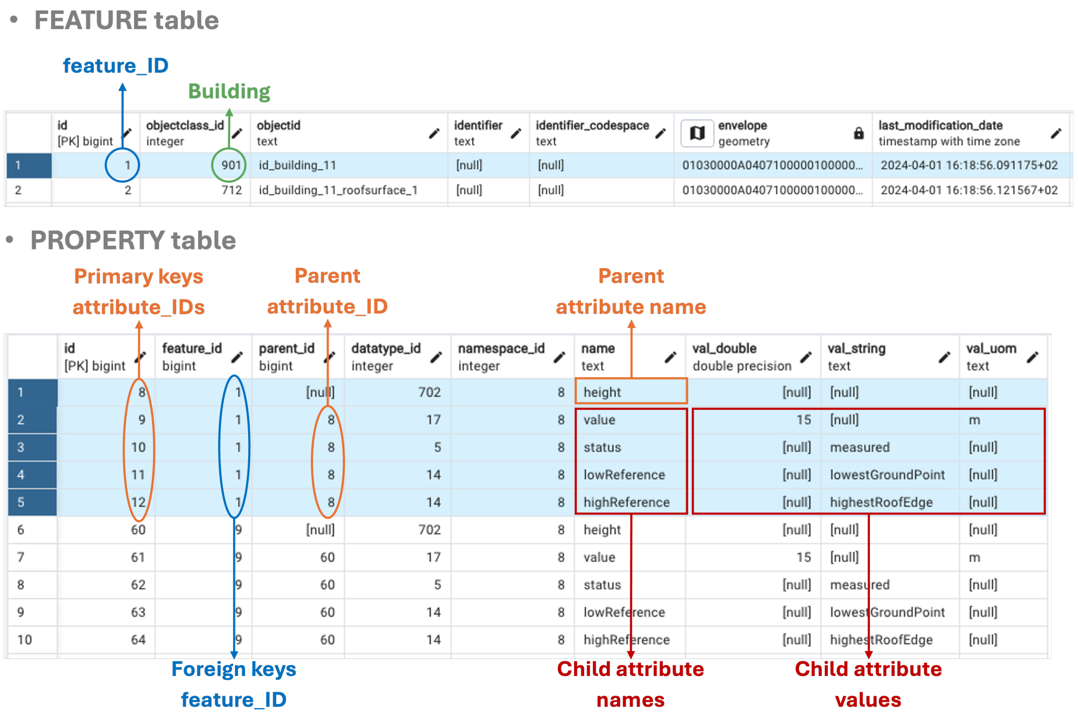

«DataType»
Height
«Property»
+ highReference: ElevationReferenceValue
+ lowReference: ElevationReferenceValue
+ status: HeightStatusValue
+ value: Length

Figure 4.7.: CityGML v.3.0 Package UML diagram - Construction module, DataType "Height" (Figure from [20])



• FEATURE table

	id [PK] bigint	objectclass_id integer	objectid text	identifier text	identifier_codespace text	envelope geometry	last_modification_date timestamp with time zone
1	1	901	id_building_11	[null]	[null]	01030000A0407100000100000...	2024-04-01 16:18:56.091175+02
2	2	712	id_building_11_roofsurface_1	[null]	[null]	01030000A0407100000100000...	2024-04-01 16:18:56.121567+02

• PROPERTY table

	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	namespace_id integer	name text	val_double double precision	val_string text	val_uom text
1	8	1	[null]	702	8	height	[null]	[null]	[null]
2	9	1	8	17	8	value	15	[null]	m
3	10	1	8	5	8	status	[null]	measured	[null]
4	11	1	8	14	8	lowReference	[null]	lowestGroundPoint	[null]
5	12	1	8	14	8	highReference	[null]	highestRoofEdge	[null]
6	60	9	[null]	702	8	height	[null]	[null]	[null]
7	61	9	60	17	8	value	15	[null]	m
8	62	9	60	5	8	status	[null]	measured	[null]
9	63	9	60	14	8	lowReference	[null]	lowestGroundPoint	[null]
10	64	9	60	14	8	highReference	[null]	highestRoofEdge	[null]

Figure 4.8.: Nested attribute example - "height" of buildings

It is important to note that attributes with the same name may exist but belong to different types within 3DCityDB v.5.0. For example, the height attribute for buildings is stored as a nested attribute, identified by data type number 702 (”Height” type) in the PROPERTY table. In contrast, the height attribute for solitary vegetation objects (trees) is stored as an inline attribute, represented by data type number 17 (”Measure” Type).

To differentiate between attribute data types, the ATTRIBUTE_DATATYPE_LOOKUP table established during the `qgis_pkg` installation (see Section 4.1) is referenced for checking the mapping rules. This lookup table provides essential information for identifying attribute types and determining their respective value columns, which enables the PL/pgSQL functions to retrieve feature attributes from the database accurately.

4.3.1.2. Multiplicity

The attribute multiplicity is the minimal and maximal number of occurrences of the attribute per object [3]. This is indicated by the bracketed numbers, ”[0..*]”, following the attribute names in the CityGML UML diagrams. For example, a building can have no ”name” attribute or multiple ”name” attributes as defined in its top-level class, ”AbstractFeature”. Similarly, the ”function” attributes of a building have a zero-to-many multiplicity, meaning a building can have no entries or multiple entries of building functions according to the specification in the ”AbstractBuilding” class (Figure 4.9).

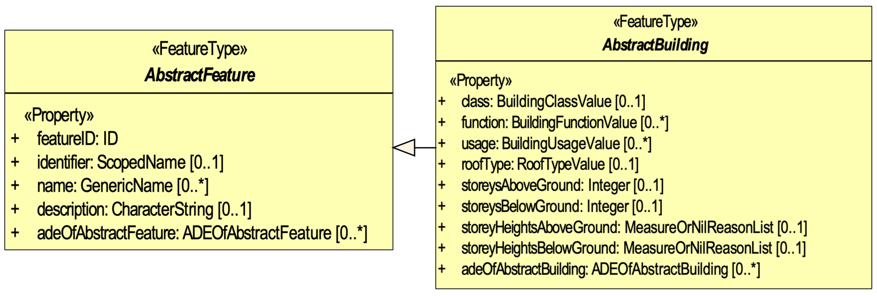
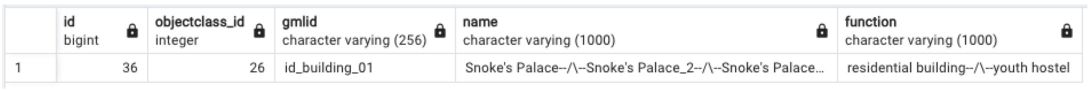

«FeatureType»
AbstractFeature
«Property»
+ featureID: ID
+ identifier: ScopedName [0..1]
+ name: GenericName [0..*]
+ description: CharacterString [0..1]
+ adeOfAbstractFeature: ADEOfAbstractFeature [0..*]
«FeatureType»
AbstractBuilding
«Property»
+ class: BuildingClassValue [0..1]
+ function: BuildingFunctionValue [0..*]
+ usage: BuildingUsageValue [0..*]
+ roofType: RoofTypeValue [0..1]
+ storeysAboveGround: Integer [0..1]
+ storeysBelowGround: Integer [0..1]
+ storeyHeightsAboveGround: MeasureOrNilReasonList [0..1]
+ storeyHeightsBelowGround: MeasureOrNilReasonList [0..1]
+ adeOfAbstractBuilding: ADEOfAbstractBuilding [0..*]

Figure 4.9.: Example of the attribute multiplicity - ”AbstractBuilding” class (Figure adapted from [20])

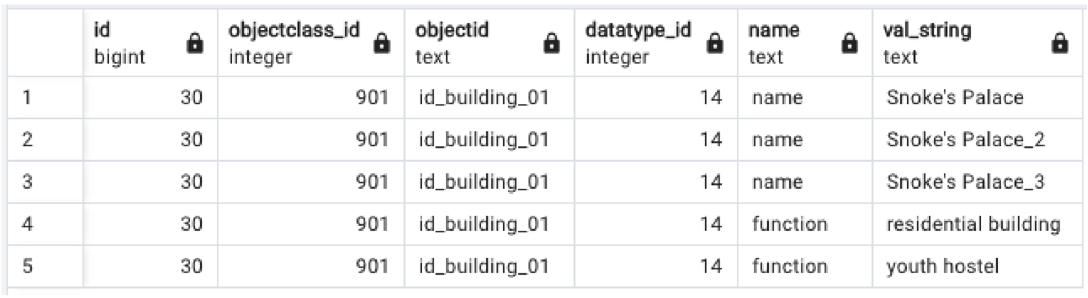
The data encoding of feature attribute multiplicity changes significantly between 3DCityDB v.4.x and 3DCityDB v.5.0. Figure 4.10 illustrates how the “name” and “function” attributes of a building feature with the “gmlid” of “id_building_01” are encoded in 3DCityDB v.4.x. In this version, “name” and “function” are flattened and stored within a single row. The “name” attribute, with a multiplicity of three, is stored as “Snoke’s Palace–/\–Snoke’s Palace_2–/\–Snoke’s Palace_3,” separated by the delimiter “–/\–.” Similarly, the “function” attribute, with a multiplicity of two, is stored as “residential building–/\–youth hostel,” using the same delimiter. Since attributes are flattened in this way, they can be directly joined with feature geometry views for GIS layer creation.



	id bigint	objectclass_id integer	gmlid character varying (256)	name character varying (1000)	function character varying (1000)
1	36	26	id_building_01	Snoke's Palace--/\--Snoke's Palace_2--/\--Snoke's Palace...	residential building--/\--youth hostel

Figure 4.10.: Attribute multiplicity encoding example - the “name” and “ function” of a building in 3DCityDB v.4.x

In 3DCityDB v.5.0, feature attributes are stored using the EAV model. The “name” and “function” attributes of the same building are encoded vertically into multiple rows. Each row contains the attribute data types followed by the attribute names, and the attribute values are stored in the `val_string` column according to the data type mapping rules (Figure 4.11). The attributes stored following the EAV model offer the advantage of direct access to attribute values, facilitating value editing. However, this method of attribute encoding cannot be directly joined with feature geometry views. Consequently, additional table operations are necessary to flatten (or “linearise”) the attributes from the query results.



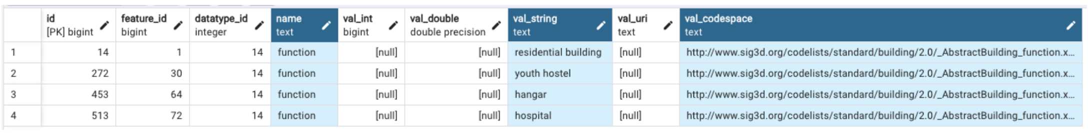
	id bigint	objectclass_id integer	objectid text	datatype_id integer	name text	val_string text
1	30	901	id_building_01	14	name	Snoke's Palace
2	30	901	id_building_01	14	name	Snoke's Palace_2
3	30	901	id_building_01	14	name	Snoke's Palace_3
4	30	901	id_building_01	14	function	residential building
5	30	901	id_building_01	14	function	youth hostel

Figure 4.11.: Attribute multiplicity encoding example - the “name” and “ function” of a building in 3DCityDB v.5.x

4.3.1.3. Attribute Value Columns

Table 4.4 provides an overview of the eighteen attribute value columns in the PROPERTY table of 3DCityDB v.5.0, as introduced in Section 2.2.2. Based on the attribute data types, the corresponding mapping rules specify how their values are stored across these eighteen columns. Feature attributes can have their values stored in multiple columns. For example, the ”function” attributes of buildings are mapped into the `val_string` column, which shows the actual functions like ”residential building” or ”hostel”. Additionally, the reference links of the codelist for retrieving these function values are mapped into the `val_codespace` column as shown in Figure 4.12.

Checking where the attribute values are stored is necessary for the attribute flattening process, as it determines the target columns that store the actual attribute values and how these values are aggregated to construct composite types required for flattening (linearising) feature attributes if the crosstab is in use. The attribute value column information is included in the ATTRIBUTE_DATATYPE_LOOKUP table for referencing.



	id [PK] bigint	feature_id bigint	datatype_id integer	name text	val_int bigint	val_double double precision	val_string text	val_uri text	val_codespace text
1	14	1	14	function	[null]	[null]	residential building	[null]	http://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_function.x...
2	272	30	14	function	[null]	[null]	youth hostel	[null]	http://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_function.x...
3	453	64	14	function	[null]	[null]	hangar	[null]	http://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_function.x...
4	513	72	14	function	[null]	[null]	hospital	[null]	http://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_function.x...

Figure 4.12.: Attribute value storage example - the “function” of buildings in 3DCityDB v.5.0

No	Value Column Name	Storage Value	PostgreSQL Data Type
1	val_int	integer	bigint
2	val_double	float number	double precision
3	val_string	string	text
4	val_timestamp	date	time with time zone
5	val_uri	Uniform Resource Identifier (URI)	text
6	val_codespace	source links of codelists or enumerations	text
7	val_uom	Unit of Measure (UoM) related to the numeric attributes	text
8	val_array	numeric array	json
9	val_lod	LoD value in string	text
10	val_geometry_id	foreign key IDs for joining the GEOMETRY_DATA table	bigint
11	val_implicitgeom_id	foreign key IDs for joining the IMPLICIT_GEOMETRY table	bigint
12	val_implicitgeom_refpoint	point geometries for feature implicit representation	geometry
13	val_appearance_id	foreign key IDs for joining the APPEARANCE table	bigint
14	val_address_id	foreign key IDs for joining the ADDRESS table	bigint
15	val_feature_id	foreign key IDs for joining FEATURE table	bigint
16	val_relation_type	integer representing a specific feature property relation type	integer
17	val_content	ADE-related attribute values	text
18	val_content_mime_type	ADE-related attribute values	text

Table 4.4.: Overview of the value columns in 3DCityDB v.5.0 PROPERTY table

After identifying the key factors to query and flatten feature attributes, the distinct feature attributes of the existing classes within the target 3DCityDB v.5.0 schema can be checked. To store the distinct attributes of each feature class, the FEATURE_ATTRIBUTE_METADATA table should include at least the following columns:

1. **Schema Name**: Identifier of the target 3DCityDB schemas.
2. **Objectclass_id**: Identifier of the feature classes.
3. **Attribute Name**: Represents the available feature attributes in the 3DCityDB v.5.0 PROPERTY table. Example entries could be the "description" or "function" of buildings.
4. **Attribute Data Type**: Identifier of the feature attribute data type implicitly indicates whether the attribute is inline or nested. Since attributes with the same name could be mapped to different types in the 3DCityDB v.5.0 encoding, it is necessary to include the attribute data types within the attribute record.

The attribute multiplicity and value columns are checked and updated in the FEATURE_ATTRIBUTE_METADATA table when the users select it. This approach avoids repetitively referencing the PROPERTY table, particularly for large datasets, to enhance query performance for checking feature attributes.

Similar to the feature geometry process, extent selection is enabled with the bounding box specifications. Users can either perform a complete scan or an optional extent selection scan on the target dataset. The FEATURE_ATTRIBUTE_METADATA table will be populated with the available distinct feature attributes of the existing classes. This metadata table then serves as a reference for users to create feature attribute views.

4.3.2. Create Views or Materialized Views for Feature Attributes

Users can select the desired feature attributes within the target database schema via the FEATURE_GEOMETRY_METADATA table. However, since feature attributes are stored using the EAV model in 3DCityDB v.5.0, these selected attributes require flattening to join with feature geometry views for creating GIS layers. Section 4.3.2.1 introduces the crosstab function supported by PostgreSQL, which is used to flatten feature attributes regarding specific attribute encoding in 3DCityDB v.5.0.

Section 4.3.2.2 to Section 4.3.2.5 elaborate on the methods applied to flatten (linearise) four different attribute classes based on the data type and multiplicity of the attribute: "Inline"-"Single", "Inline-Multiple", "Nested-Single", and "Nested-Multiple". The PL/pgSQL functions developed in this research first check the attribute classes by looking up the collect and flatten the feature attributes according to these four attribute classes and store the flattened results into views or materialized views.

4.3.2.1. Crosstab Function in PostgreSQL

The crosstab function included in the PostgreSQL `tablefunc` extension is introduced to transpose the attributes from rows to columns when each unique feature has its attributes stored in multiple rows within the PROPERTY table.

According to [36], the crosstab function is used to create "pivot" displays, where data is horizontally listed across columns rather than vertically down rows. This function requires SQL statement as a text parameter, which generates a source table with three essential columns: one for row name, another for category, and the third for values. The column header names and types of the crosstab function must be defined in the FROM clause of the calling SELECT statement.

As illustrated in the Listing 4.3 and Figure 4.13, the row names from the source table represent unique entries. These must be defined as the first column header in the crosstab FROM clause, with the same data type as in the source table (in this case, a primary key of type big integer). Following the row name column, the crosstab FROM clause includes several value columns. The headers for these columns are derived from the category values, and their data type must match the data type of the value column in the source table (in this case, it is set to the text type). This ensures that the categorised value columns in the resulting table correctly store values from the source table based on their data types.

```sql
SELECT *
FROM CROSSTAB('
    SELECT
        Row_name,
        Category,
        Values
    FROM source_table') -- SQL statement as the source table
        AS ct(row_name bigint, category_1 text, category_2 text);
```

Listing 4.3: Crosstab template

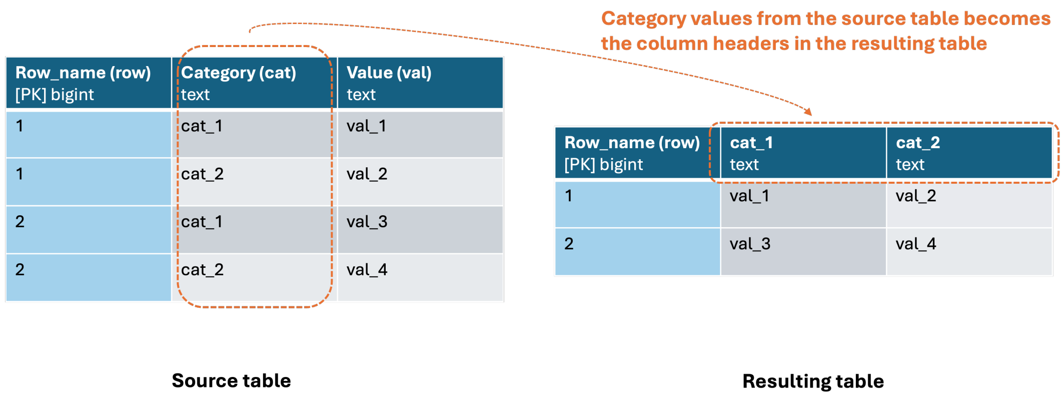


Figure 4.13.: Visual reference of the crosstab function

The crosstab function generates one output row for each group of consecutive rows sharing the same row name value from the source table. It populates the value columns in the resulting table from left to right with the value fields from these rows. If a group has fewer rows than the number of output value columns (Listing 4.4), the remaining columns are filled with nulls. Conversely, if a group has more rows than the output value columns (Listing 4.5), the extra input rows are ignored (Figure 4.14).

```sql
SELECT * FROM CROSSTAB('source table SQL statement')
    AS ct(row_name bigint, category_1 text);
```

Listing 4.4: Crosstab template with fewer value columns

```sql
SELECT * FROM CROSSTAB('source table SQL statement')
    AS ct(row_name bigint, category_1 text, category_2 text, , category_3 text);
```

Listing 4.5: Crosstab template with more value columns

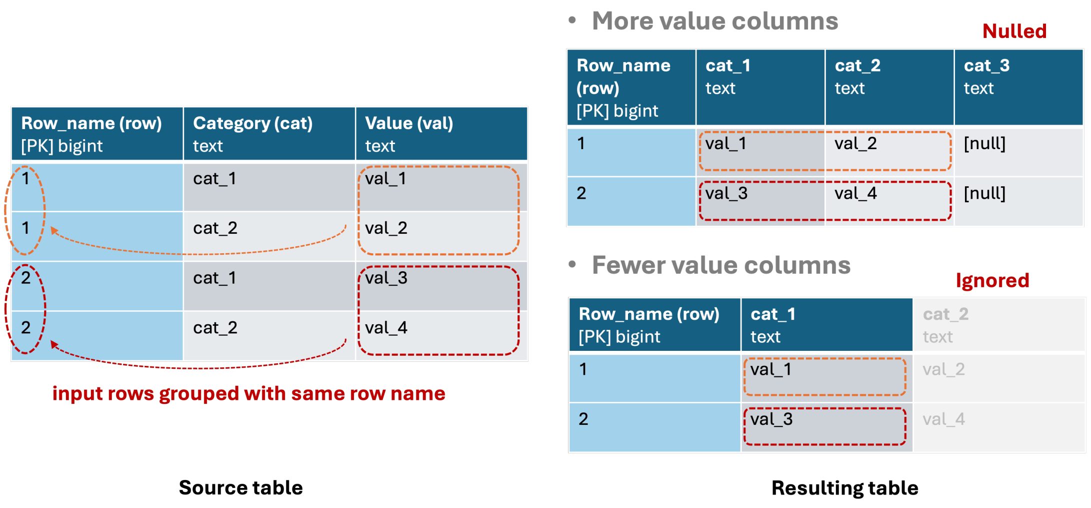


Figure 4.14.: Visual reference of the crosstab function with more or fewer value columns

4.3.2.2. Inline-Single Attribute Class

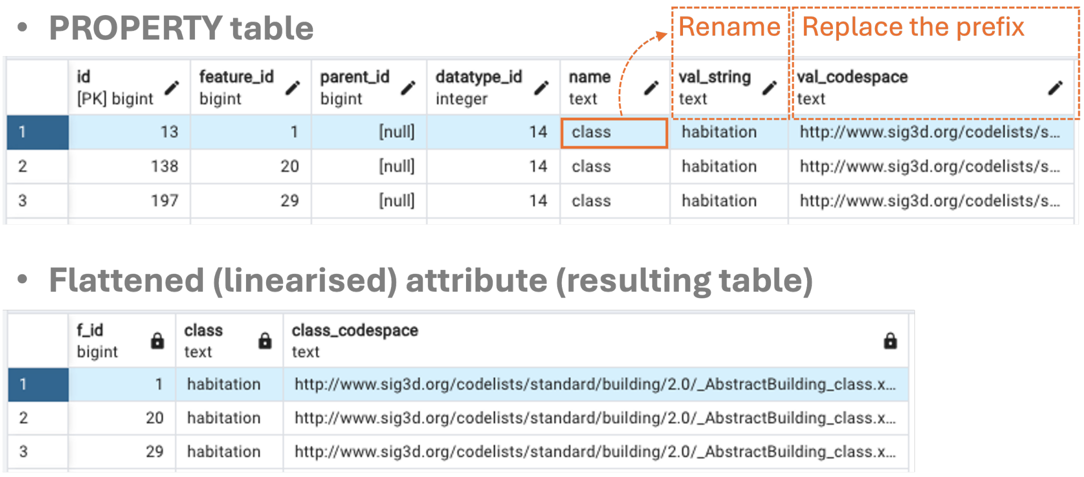
The Inline-Single class represents the simplest scenario for flattening feature attributes. Since this class of attributes is already stored following the SFS model, it only requires renaming the value column headers using the inline attribute names. The following principles apply based on the number of value columns:

1. **One value column**: Rename the single-column header with the attribute name.
2. **Multiple value columns**: Rename the first value column header with the attribute name. For the rest of the value column headers, replace the prefix "val_" with the attribute name, as these are supplementary columns.

Listing 4.6 and Figure 4.15 provide examples of how to query and flatten the Inline-Single attribute. The "class" of buildings serves as an example of an inline attribute with a multiplicity of one. This means that each building has a single "class" attribute stored in one row within the PROPERTY table, with values across `val_string` and `val_codespace` columns. According to the principles, the header of the first column is renamed to "class", and the prefixes of the remaining value columns are replaced with the attribute name. The implementation and the query template for the Inline-Single attributes are provided in Section 5.4.3.1.

```sql
SELECT
    f.id AS f_id,
    -- Rename 1st value column to the attribute name
    p.val_string AS "class",
    -- Replace the value column prefix with the attribute name
    p.val_codespace AS "class_codespace"
FROM citydb.feature AS f
    INNER JOIN citydb.property AS p ON f.id = p.feature_id
        AND f.objectclass_id = 901 -- Building class
        AND p.name = 'class'; -- Target attribute
```

Listing 4.6: Flatten query example of Inline-Single attributes: ”class” of buildings



	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	name text	val_string text	val_codespace text
1	13	1	[null]	14	class	habitation	http://www.sig3d.org/codelists/s...
2	138	20	[null]	14	class	habitation	http://www.sig3d.org/codelists/s...
3	197	29	[null]	14	class	habitation	http://www.sig3d.org/codelists/s...

	f_id bigint	class text	class_codespace text
1	1	habitation	http://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_class.x...
2	20	habitation	http://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_class.x...
3	29	habitation	http://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_class.x...

Figure 4.15.: Visual reference of flattening Inline-Single attribute: ”class” of buildings

4.3.2.3. Inline-Multiple Attribute Class

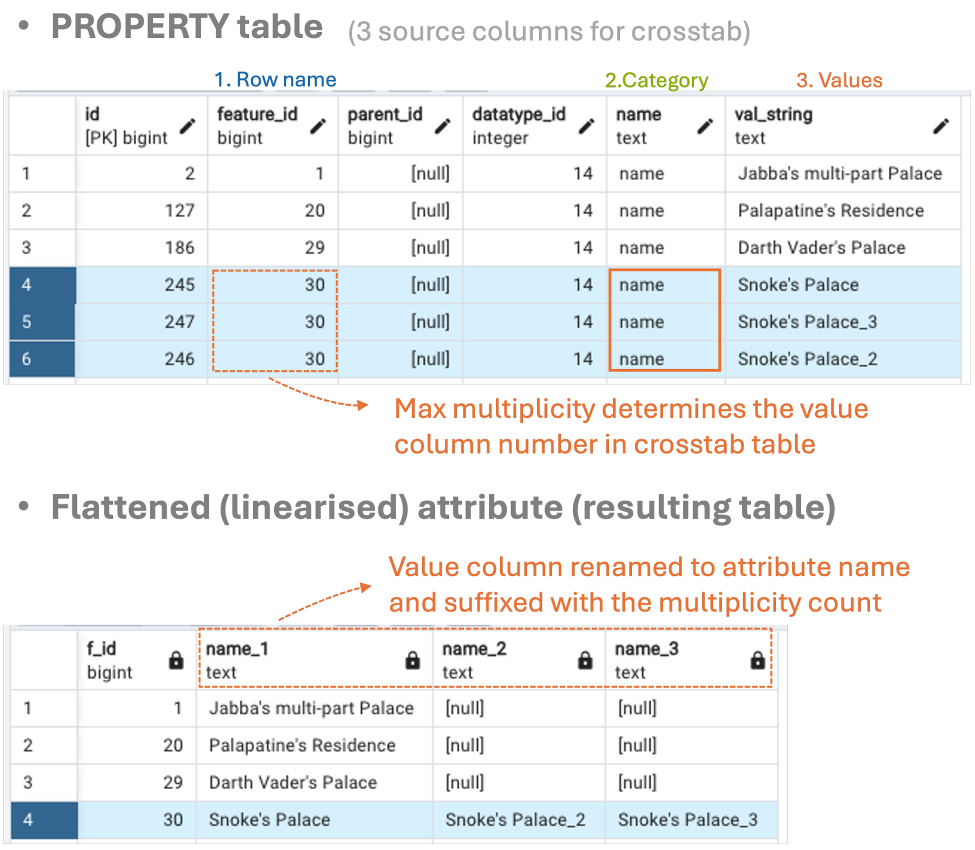
If a feature inline attribute has a multiplicity greater than one, it falls under the Inline-Multiple class. In this scenario, the crosstab function is required to flatten the attributes, as a unique feature can have multiple entries of an attribute stored across rows within the PROPERTY table. The crosstab function for flattening Inline-Multiple attributes follows two principles based on the number of value columns:

1. **One value column**: Listing 4.7 and Figure 4.16 show an example of querying and flattening an Inline-Multiple attribute with one value column, which is the ”name” of buildings. In the PROPERTY table, the three source columns for the crosstab are labelled in blue, green, and orange, respectively. The query of these source columns is used as the SQL text input, enclosed by the `$BODY$` tags for the crosstab function.

   Note that the maximum multiplicity of the selected attribute determines the number of value columns in the crosstab `FROM` clause, as it requires a corresponding number of columns to hold the transposed values. Therefore, if an Inline-Multiple attribute is selected by the users, a full scan of the dataset within the user-selected extent should be performed first to check its maximum multiplicity. The value columns in the resulting table are then renamed to the attribute name suffixed with the multiplicity count.

```
SELECT
    f_id AS f_id,
    ct.name_1 AS name_1,
    ct.name_2 AS name_2,
    ct.name_3 AS name_3
FROM CROSSTAB($BODY$
    SELECT
        f.id AS f_id,   -- 1. row name
        p.name,         -- 2. category
        p.val_string     -- 3. values
    FROM citydb.feature AS f
        INNER JOIN citydb.property AS p ON (f.id = p.feature_id
            AND f.objectclass_id = 901)
    WHERE p.name = 'name'
    ORDER BY f_id, p.id ASC $BODY$)
    -- Types specified based on the 1 and 3 columns from the source table
    AS ct(f_id bigint, "name_1" text,"name_2" text,"name_3" text);
```

Listing 4.7: Flatten query example of Inline-Multiple attributes with one value column: "name" of buildings


• PROPERTY table (3 source columns for crosstab)
1. Row name
2.Category
3. Values
Max multiplicity determines the value column number in crosstab table
• Flattened (linearised) attribute (resulting table)
Value column renamed to attribute name and suffixed with the multiplicity count

	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	name text	val_string text
1	2	1	[null]	14	name	Jabba's multi-part Palace
2	127	20	[null]	14	name	Palapatine's Residence
3	186	29	[null]	14	name	Darth Vader's Palace
4	245	30	[null]	14	name	Snoke's Palace
5	247	30	[null]	14	name	Snoke's Palace_3
6	246	30	[null]	14	name	Snoke's Palace_2

	f_id bigint	name_1 text	name_2 text	name_3 text
1	1	Jabba's multi-part Palace	[null]	[null]
2	20	Palapatine's Residence	[null]	[null]
3	29	Darth Vader's Palace	[null]	[null]
4	30	Snoke's Palace	Snoke's Palace_2	Snoke's Palace_3

Figure 4.16.: Visual reference of flattening Inline-Multiple attributes with one value column: "name" of buildings

2. **Multiple value columns**: Similar to the one-value column case, the maximum attribute

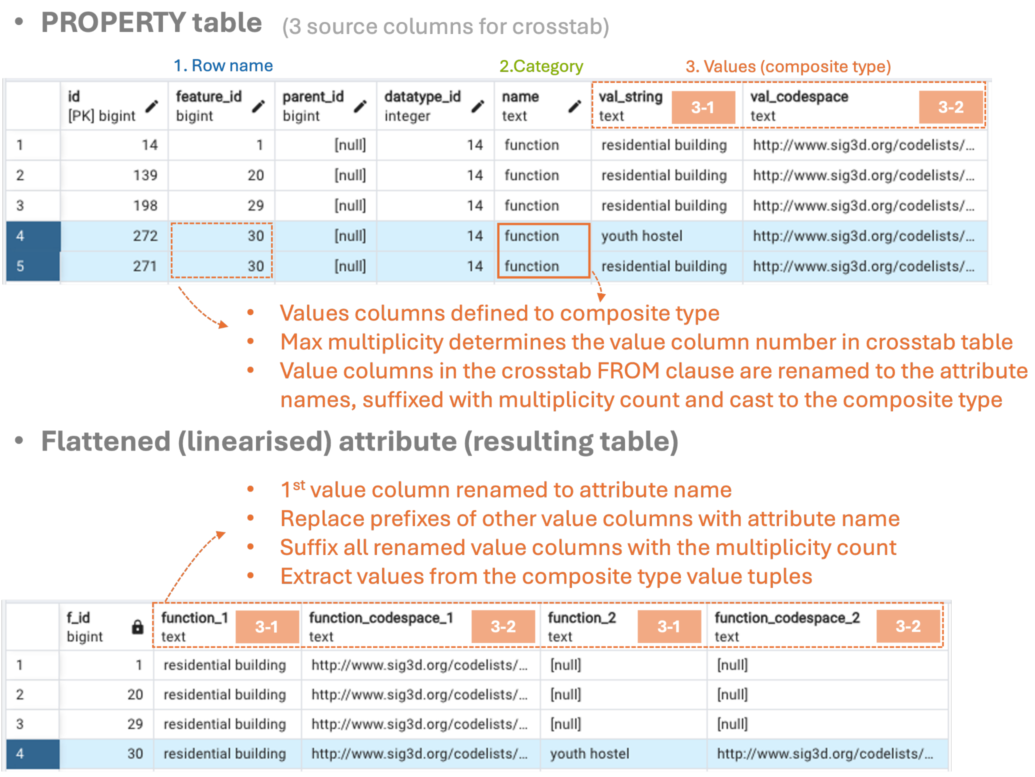
multiplicity determines the number of value columns in the crosstab FROM clause. However, since the crosstab function only accepts one value column from the source table, it is necessary to combine all columns storing attribute values in the PROPERTY table into a composite type. This ensures that the attribute values are stored in a single column for the crosstab function, allowing the actual values to be extracted from the composite-type tuples after flattening.

In the example shown in Listing 4.8 and Figure 4.17, the composite type named ”citydb_901_function_ct” is defined first, composed of two text values. In the source table of the crosstab function, the val_string and val_codespace columns are combined and cast to the composite type, ensuring only one value column for the crosstab function. The maximum attribute multiplicity, which is two in this case, determines the number of value columns in the resulting table, all cast to the composite type. Finally, the actual values are extracted from the resulting table value columns using dot notation, with the composite type composed of two text values: the first from the val_string column and the second from the val_codespace column.

The naming convention for composite types in this research follows a specific format: the target dataset schema name is combined with the objectclass_id, attribute name, and a ”ct” suffix, using underscores as delimiters. Composite types are defined when users select attributes with a multiplicity greater than one and have values stored across multiple value columns in the PROPERTY table. When created, the composite type name is stored in the FEATURE_ATTRIBUTE_METADATA table for management and reference. The implementation and the query template for the Inline-Multiple attributes are provided in Section 5.4.3.2.

```sql
-- Define composite type for holding value in tuples from the source table
DROP TYPE IF EXISTS "citydb_901_function_ct";
CREATE TYPE "citydb_901_function_ct" AS (val_string text,val_codespace text);
-- Flattening attributes using composite type
SELECT
    f_id AS f_id,
    -- Extract values from composite-type tuples
    (function_1).val_string AS "function_1",
    (function_1).val_codespace AS "function_codespace_1",
    (function_2).val_string AS "function_2",
    (function_2).val_codespace AS "function_codespace_2"
FROM CROSSTAB($BODY$
    SELECT
        f.id AS f_id,  -- 1. row name
        p.name,        -- 2. category
        (p.val_string,p.val_codespace)::"citydb_901_function_ct" -- 3. values
    FROM citydb.feature AS f
        INNER JOIN citydb.property AS p ON (f.id = p.feature_id
            AND f.objectclass_id = 901)
    WHERE p.name = 'function'
    ORDER BY f_id, p.id ASC $BODY$)
    -- Max multiplicity decides the number of the value column
    -- Source value columns cast to the composite type
    AS ct(f_id bigint,
        function_1 "citydb_901_function_ct",
        function_2 "citydb_901_function_ct");
```

Listing 4.8: Flatten query example of Inline-Multiple attributes with multiple value columns: ”function” of buildings



- **PROPERTY table** (3 source columns for crosstab)

	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	name text	val_string text	val_codespace text
1	14	1	[null]	14	function	residential building	http://www.sig3d.org/codelists/...
2	139	20	[null]	14	function	residential building	http://www.sig3d.org/codelists/...
3	198	29	[null]	14	function	residential building	http://www.sig3d.org/codelists/...
4	272	30	[null]	14	function	youth hostel	http://www.sig3d.org/codelists/...
5	271	30	[null]	14	function	residential building	http://www.sig3d.org/codelists/...

- Values columns defined to composite type
- Max multiplicity determines the value column number in crosstab table
- Value columns in the crosstab FROM clause are renamed to the attribute names, suffixed with multiplicity count and cast to the composite type

- **Flattened (linearised) attribute (resulting table)**

- 1st value column renamed to attribute name
- Replace prefixes of other value columns with attribute name
- Suffix all renamed value columns with the multiplicity count
- Extract values from the composite type value tuples

	f_id bigint	function_1 text	function_codespace_1 text	function_2 text	function_codespace_2 text
1	1	residential building	http://www.sig3d.org/codelists/...	[null]	[null]
2	20	residential building	http://www.sig3d.org/codelists/...	[null]	[null]
3	29	residential building	http://www.sig3d.org/codelists/...	[null]	[null]
4	30	residential building	http://www.sig3d.org/codelists/...	youth hostel	http://www.sig3d.org/codelists/...

Figure 4.17.: Visual reference of flattening Inline-Multiple attribute with multiple value columns: "function" of buildings

4.3.2.4. Nested-Single Attribute Class

The Nested-Single class stands for the nested attributes with single entries. An entry of nested attributes consists of one parent inline attribute followed by several inline child attributes connected via the `parent_ids`, and the values are usually stored across multiple value columns in the PROPERTY table. Listing 4.9 and Figure 4.18 provide an example of querying and flattening a Nested-Single attribute with multiple value columns, specifically the "height" of buildings. A single "height" attribute entry for buildings consists of four child attributes, which are queried by repetitively cross-referencing the PROPERTY table.

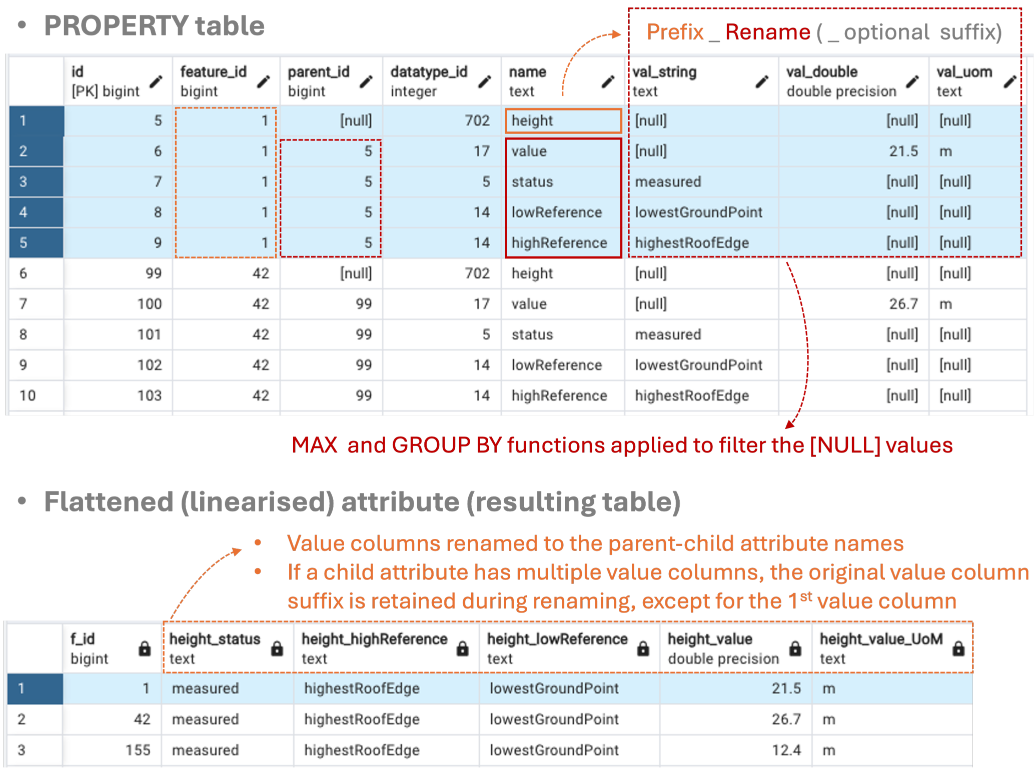
The first join queries the "Building" class and the "height" attribute, while the second join sorts out the child attributes of "height". Since the child attributes are all inline attributes, their value columns can be identified by referencing the ATTRIBUTE_DATATYPE_LOOKUP table. The `CASE` function handles different child attributes by searching for values in the corresponding value columns, and the column headers in the resulting table are renamed to the parent-child attribute name combinations.

If a child attribute has multiple value columns, the suffixes starting from the second value column are retained and added after the rename. For instance, the "value" of "height" has two value columns: `val_double` and `val_uom`. The first column is renamed to "height_value", while the second column is renamed to "height_value_UoM" to indicate it is the supplementary column for "value", representing the Unit of Measure (UoM).

Finally, the MAX and GROUP BY functions are applied to filter out NULL values in the resulting table, ensuring the flattened result is inline. The implementation and the query template for the Nested-Single attributes are provided in Section 5.4.3.3.

```sql
SELECT
    f.id AS f_id,
    -- MAX, GROUP BY functions are used to filter the [NULL] values
    MAX(CASE WHEN p1.name = 'status'
        THEN p1.val_string END) AS "height_status",
    MAX(CASE WHEN p1.name = 'highReference'
        THEN p1.val_string END) AS "height_highReference",
    MAX(CASE WHEN p1.name = 'lowReference'
        THEN p1.val_string END) AS "height_lowReference",
    MAX(CASE WHEN p1.name = 'value'
        THEN p1. val_double END) AS "height_value",
    MAX(CASE WHEN p1.name = 'value'
        THEN p1.val_uom END) AS "height_value_UoM"
FROM citydb.feature AS f
    -- First PROPERTY table join for the querying the parent attribute
    INNER JOIN citydb.property AS p ON (f.id = p.feature_id
        AND f.objectclass_id = 901 AND p.name = 'height')
    -- Second PROPERTY table join for the querying the child attributes
    INNER JOIN citydb.property AS p1 ON p.id = p1.parent_id
GROUP BY f.id;
```

Listing 4.9: Flatten query example of Nested-Single attributes: ”height” of buildings



• PROPERTY table

Prefix _ Rename (_ optional suffix)

	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	name text	val_string text	val_double double precision	val_uom text
1	5	1	[null]	702	height	[null]	[null]	[null]
2	6	1	5	17	value	[null]	21.5	m
3	7	1	5	5	status	measured	[null]	[null]
4	8	1	5	14	lowReference	lowestGroundPoint	[null]	[null]
5	9	1	5	14	highReference	highestRoofEdge	[null]	[null]
6	99	42	[null]	702	height	[null]	[null]	[null]
7	100	42	99	17	value	[null]	26.7	m
8	101	42	99	5	status	measured	[null]	[null]
9	102	42	99	14	lowReference	lowestGroundPoint	[null]	[null]
10	103	42	99	14	highReference	highestRoofEdge	[null]	[null]

MAX and GROUP BY functions applied to filter the [NULL] values

• Flattened (linearised) attribute (resulting table)

• Value columns renamed to the parent-child attribute names
• If a child attribute has multiple value columns, the original value column suffix is retained during renaming, except for the 1st value column

	f_id bigint	height_status text	height_highReference text	height_lowReference text	height_value double precision	height_value_UoM text
1	1	measured	highestRoofEdge	lowestGroundPoint	21.5	m
2	42	measured	highestRoofEdge	lowestGroundPoint	26.7	m
3	155	measured	highestRoofEdge	lowestGroundPoint	12.4	m

Figure 4.18.: Visual reference of flattening Nested-Single attribute: ”height” of buildings

4.3.2.5. Nested-Multiple Attribute Class

If a unique feature has a nested attribute with a multiplicity greater than one, it belongs to the Nested-Multiple class. In this scenario, the crosstab function used for flattening Inline-Multiple attributes (Section 4.3.2.3) is combined with the renaming approach for Nested-Single attributes (Section 4.3.2.4). Listing 4.10 and Figure 4.19 provide an example of the ”height” of buildings with multiple entries.

The values of a building’s ”height” are stored across three columns: `val_string`, `val_double`, and `val_uom`. A composite type named ”citydb_901_height_ct” is created first following the composite type naming convention to combine these three columns into a single value column for the crosstab function. The SQL query text used as the input for flattening is identical to Listing 4.9. The FEATURE table is joined with the PROPERTY table twice to query all child attributes of the buildings’ ”height”, selecting only `feature_id`, `name`, and the combined values column in the composite type to satisfy the three elemental columns for the crosstab function.

Similar to the Inline-Multiple attribute flattening approach, a full scan of the dataset within the user-selected extent is performed first to check the maximum multiplicity of the target attribute, which is two in this case. The maximum attribute multiplicity determines the number of value columns in the crosstab `FROM` clause. Considering each entry of the building’s ”height” has four child attributes, the value columns in the crosstab `FROM` clause are set to the child attribute names suffixed with the multiplicity counts, and all columns are cast to the composite type.

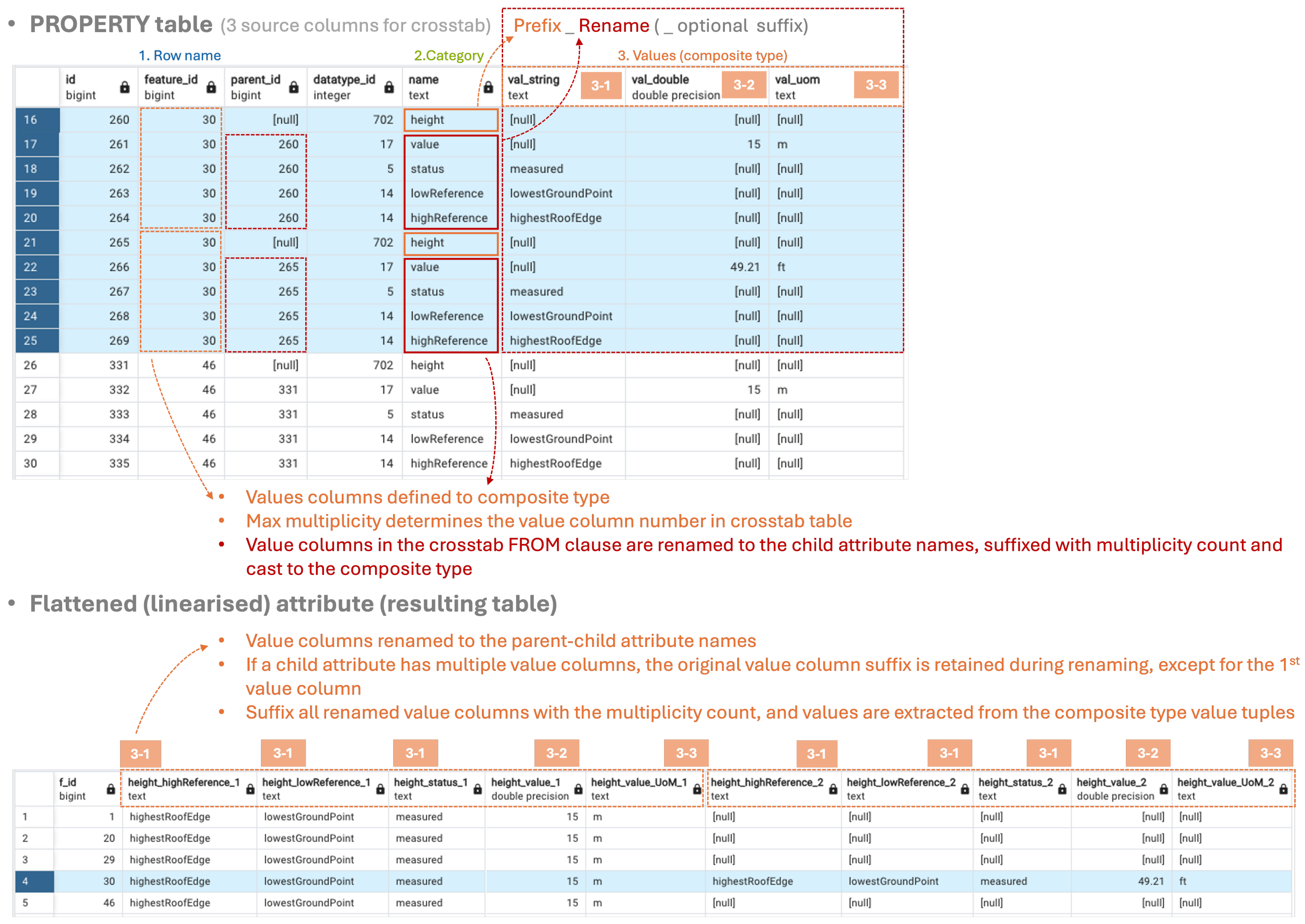
Finally, in the `SELECT` clause of the crosstab function, the corresponding child attribute value columns are checked from the ATTRIBUTE_DATATYPE_LOOKUP table, and the actual values from the crosstab result are extracted back using dot notation. The value columns holding the extracted values are renamed by the parent-child attribute name combinations suffixed with the multiplicity count. The implementation and the query template for the Nested-Multiple attributes are provided in Section 5.4.3.4.

```sql
-- Define composite type for holding value in tuples from the source table
DROP TYPE IF EXISTS "citydb_901_height_ct";
CREATE TYPE "citydb_901_height_ct"
    AS (val_string text, val_double double precision, val_uom text);
-- Flattening attributes using composite type
SELECT
    f_id AS f_id,
    -- Extract values from composite-type tuples
    (highReference_1).val_string    AS "height_highReference_1",
    (lowReference_1).val_string     AS "height_lowReference_1",
    (status_1).val_string           AS "height_status_1",
    (value_1). val_double           AS "height_value_1",
    (value_1).val_uom               AS "height_value_UoM_1",
    (highReference_2).val_string    AS "height_highReference_2",
    (lowReference_2).val_string     AS "height_lowReference_2",
    (status_2).val_string           AS "height_status_2",
    (value_2). val_double           AS "height_value_2",
    (value_2).val_uom               AS "height_value_UoM_2"
FROM CROSSTAB($BODY$
    SELECT
        f.id AS f_id, -- 1. row name
        p1.name,      -- 2. category
        -- 3. values
        (p1.val_string,p1. val_double ,p1.val_uom)::"citydb_901_height_ct"
    FROM citydb.feature AS f
        INNER JOIN citydb.property AS p ON (f.id = p.feature_id
            AND f.objectclass_id = 901 AND p.name = 'height')
        INNER JOIN citydb.property AS p1 ON p.id = p1.parent_id
    ORDER BY f.id, p.id, p1.name ASC $BODY$)
    -- Max multiplicity decides the number of the value column
    -- Source value columns cast to the composite type
    AS ct(f_id bigint,
        highReference_1 "citydb_901_height_ct",
        lowReference_1  "citydb_901_height_ct",
        status_1        "citydb_901_height_ct",
        value_1         "citydb_901_height_ct",
        highReference_2 "citydb_901_height_ct",
        lowReference_2  "citydb_901_height_ct",
        status_2        "citydb_901_height_ct",
        value_2         "citydb_901_height_ct");
```

Listing 4.10: Flatten query example of Nested-Multiple attributes: ”height” of buildings

To conclude, four attribute classes: Inline-Single, Inline-Multiple, Nested-Single, and Nested-Multiple, are defined for flattening (linearising) feature attributes from the data encoded in 3DCityDB v.5.0. The query examples demonstrate the case results of the PL/pgSQL functions for attribute flattening. The flattened (linearised) attribute can then be joined with the user-selected feature geometry views to create layers that comply with the SFS model.

Unlike the attribute query approach of 3DCityDB v.4.x, where general and specific attributes are already flattened and directly stored as views for GIS layer creation, with generic attributes associated as sub-tables, 3DCityDB v.5.0 flattens all attributes. This allows users to select desired attributes to join with geometry views individually. Given the complexity of attribute flattening and the size of the PROPERTY table, it becomes apparent that materialized views can offer a better user experience, especially when working with large datasets. The decision regarding attribute view types is further discussed in Section 4.4, as it significantly impacts the approach to GIS layer creation.


Prefix _ Rename (_ optional suffix)
1. Row name
2.Category
3. Values (composite type)
3-1
3-2
3-3

- **PROPERTY table** (3 source columns for crosstab)

	id bigint	feature_id bigint	parent_id bigint	datatype_id integer	name text	val_string text	val_double double precision	val_uom text
16	260	30	[null]	702	height	[null]	[null]	[null]
17	261	30	260	17	value	[null]	15	m
18	262	30	260	5	status	measured	[null]	[null]
19	263	30	260	14	lowReference	lowestGroundPoint	[null]	[null]
20	264	30	260	14	highReference	highestRoofEdge	[null]	[null]
21	265	30	[null]	702	height	[null]	[null]	[null]
22	266	30	265	17	value	[null]	49.21	ft
23	267	30	265	5	status	measured	[null]	[null]
24	268	30	265	14	lowReference	lowestGroundPoint	[null]	[null]
25	269	30	265	14	highReference	highestRoofEdge	[null]	[null]
26	331	46	[null]	702	height	[null]	[null]	[null]
27	332	46	331	17	value	[null]	15	m
28	333	46	331	5	status	measured	[null]	[null]
29	334	46	331	14	lowReference	lowestGroundPoint	[null]	[null]
30	335	46	331	14	highReference	highestRoofEdge	[null]	[null]

- Values columns defined to composite type
- Max multiplicity determines the value column number in crosstab table
- Value columns in the crosstab FROM clause are renamed to the child attribute names, suffixed with multiplicity count and cast to the composite type

- **Flattened (linearised) attribute (resulting table)**

- Value columns renamed to the parent-child attribute names
- If a child attribute has multiple value columns, the original value column suffix is retained during renaming, except for the 1st value column
- Suffix all renamed value columns with the multiplicity count, and values are extracted from the composite type value tuples

		3-1	3-1	3-1	3-2	3-3	3-1	3-1	3-1	3-2	3-3
	f_id bigint	height_highReference_1 text	height_lowReference_1 text	height_status_1 text	height_value_1 double precision	height_value_UoM_1 text	height_highReference_2 text	height_lowReference_2 text	height_status_2 text	height_value_2 double precision	height_value_UoM_2 text
1	1	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
2	20	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
3	29	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
4	30	highestRoofEdge	lowestGroundPoint	measured	15	m	highestRoofEdge	lowestGroundPoint	measured	49.21	ft
5	46	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]

Figure 4.19.: Visual reference of flattening Nested-Multiple attribute: "height" of buildings

4.4. GIS Layers Creation

The final step of the methodology is to join feature geometries with selected attributes to create GIS layers. As indicated in Section 4.3, feature attributes from 3DCityDB v.5.0 require flattening to comply with SFS model, meaning attribute views are generated individually based on user selection. Therefore, repetitively joining the attribute views is necessary to interact with multiple attributes.

The GIS layers consist of user-selected feature geometry materialized views (see Section 4.2.2), which are joined with one or more attributes using `LEFT JOIN` to retain feature geometries while accounting for any missing attributes. The proposed approaches for creating GIS layers are detailed below and illustrated in Figure 4.20.

1. **Geometry Materialized View joined with multiple of Attribute Views**:
   Each selected attribute is stored as a single view and joined with the target feature geometry materialized view using `LEFT OUTER JOIN` on `feature_ids`. Attribute view names should include the target schema name, `objectclass_id`, and attribute name for unique identification.
2. **Geometry Materialized View joined with multiple Attribute Materialized Views**:
   Each selected attribute is stored as a single materialized view with indices on every column, then joined with the target feature geometry materialized view using `LEFT OUTER JOIN` on `feature_ids`. The naming convention is the same as in the first approach, with another label for differentiation.
3. **Geometry Materialized View joined with one Attributes table Materialized View**:
   All selected attributes are combined into a single materialized view using `FULL JOIN` on the flattening queries, creating an integrated attribute table with indices on every column. This table is then joined with the target feature geometry materialized view using `LEFT OUTER JOIN` on `feature_ids`. The naming convention includes the target schema name, `objectclass_id`, and geometry materialized view name or another specifier for unique identification.

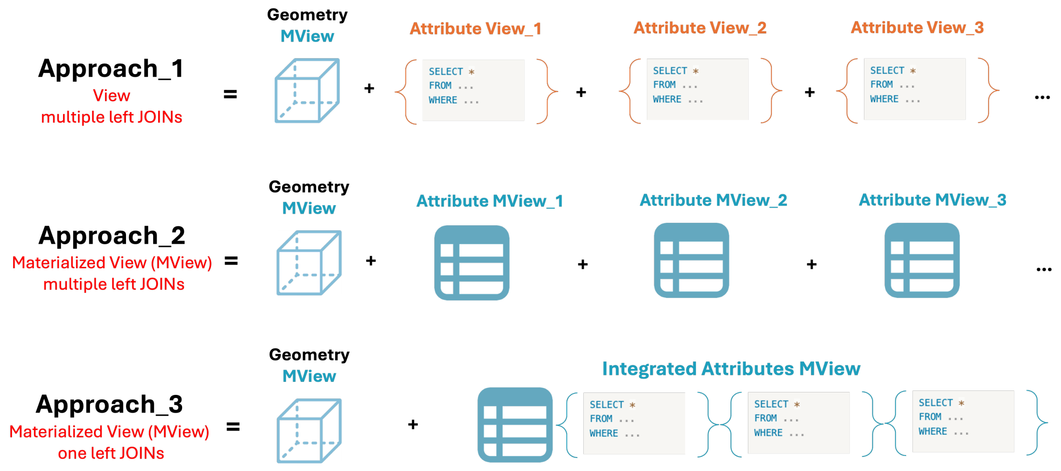


Figure 4.20.: GIS layer creation approaches

The initial approach using attribute views may result in longer query times due to the need to recompute and flatten (linearise) all selected attributes each time the GIS layers are queried. This issue becomes more pronounced with large datasets, where the recomputation process can significantly impact performance.

To address this, utilising a materialized view for storing the flattened (linearised) attributes is preferable. Materialized views precompute and store the flattened data, which improves query performance by eliminating the need for on-the-fly computation and thereby enhances query time efficiency, especially with large datasets. A query performance test is conducted to evaluate the query times for GIS layers generated by these approaches. The results of this test are discussed in Section 6.2.

The metadata of the generated GIS layer, including the selected approaches, feature geometry LoDs representations and attributes, are stored within the LAYER_METADATA table, which is elaborated in Section 6.3. The GIS layers generated from different CityGML datasets using the proposed approaches are shown in Section 6.4.

5. Implementation Part 1: Feature Geometry and Attribute Metadata Check and View Creation

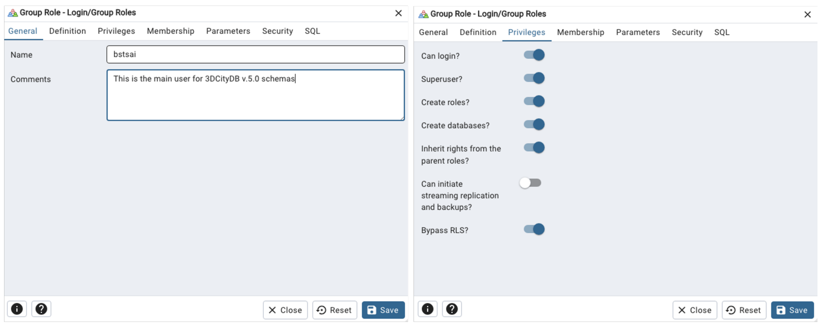
This chapter introduces the implementation to set up the `qgis_pkg` for creating and populating the FEATURE_GEOMETRY_METADATA and FEATURE_ATTRIBUTE_METADATA tables for view creation from the data stored within 3DCityDB v.5.0. The metadata tables are referenced as menus for users to create GIS layers for further interaction in QGIS. Section 5.1 introduces the structure of the PL/pgSQL function package, explaining the setup and installation. Section 5.2 briefly introduces the CityGML datasets used in this thesis. Section 5.3 focuses on preparing the feature geometry metadata table and presents the process and visualisation of the created feature geometry materialized views. Section 5.4 details preparing the feature attribute metadata table and shows the result of flattened attributes in views.

5.1. 3DCityDB and the PL/pgSQL Function Package Setup

The initial phase of importing CityGML data into the PostgreSQL database is to set up the 3DCityDB v.5.0 instance. Users can download the latest 3DCityDB tool package from the open-source repository on GitHub ([22]). The steps to create a 3DCityDB v.5.0 instance are introduced in Section 5.1.1, while Section 5.1.2 elaborates on the structure of the `qgis_pkg` and its installation on 3DCityDB v.5.0 instances.

5.1.1. 3DCityDB v.5.0 Setup

1. **Create new user in pgAdmin4**
   Open the pgAdmin4 application. In the side panel under ”Login/Group Roles,” create a new user. Set privileges for this newly created superuser by granting full access.



Name: bstsai
Comments: This is the main user for 3DCityDB v.5.0 schemas

Can login? ✓
Superuser? ✓
Create roles? ✓
Create databases? ✓
Inherit rights from the parent roles? ✓
Can initiate streaming replication and backups? ✗
Bypass RLS? ✓

Figure 5.1.: Create a new user name and grant superuser privileges in pgAdmin4

2. **Create new database and extensions**
   Create a new database under the PostgreSQL server (version 15). The following extensions are required to install the 3DCityDB v.5.0 instance, which can be achieved by executing the SQL queries shown in Listing 5.1.

```
CREATE EXTENSION IF NOT EXISTS postgis        SCHEMA public;
CREATE EXTENSION IF NOT EXISTS postgis_raster SCHEMA public;
CREATE EXTENSION IF NOT EXISTS "uuid-ossp"    SCHEMA public;
CREATE EXTENSION IF NOT EXISTS pldbgapi       SCHEMA public;
```

Listing 5.1: Create extensions for installing 3DCityDB instances

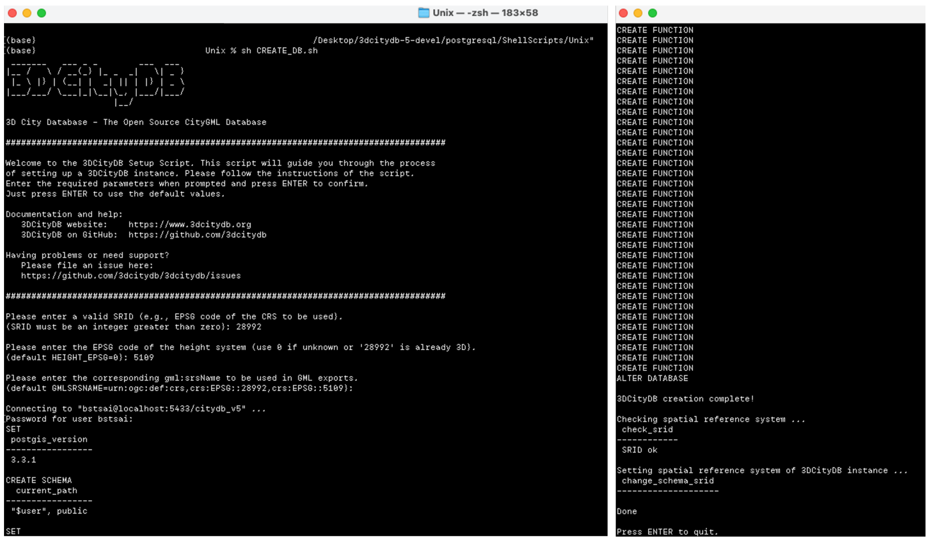
3. **Setup connection details**
   After creating a new database and setting up the necessary extensions, navigate to the ”connection detail” shell script file in the 3DCityDB tool package. Fill out the connection details, an example is given in Listing 5.2.

```
export PGBIN=/Library/PostgreSQL/16/bin
export PGHOST=localhost
export PGPORT=5433
export CITYDB=citydb_v5
export PGUSER=bstsai;
```

Listing 5.2: Setup connection detail example

4. **Create 3DCityDB schema**
   Execute the CREATE_DB shell script using the terminal to create the 3DCityDB v.5.0 schema in the same directory as the previous step. A window prompt will appear, requesting the entry of the SRID and EPSG code for the height system regarding the dataset. For example, the SRID of the ”Rijsen-Holten” dataset [37] is set to 28992, which is the projected coordinate system for the Netherlands, and the height EPSG code is set to 5109.

   After successfully creating the 3DCityDB v.5.0 schema, two schemas with the default names ”citydb” and ”citydb_pkg” are added under the newly created database. Users can create additional 3DCityDB v.5.0 schemas by executing the CREATE_SCHEMA shell script, where the coordinate settings are duplicated from the default 3DCityDB schema. The CityGML datasets can then be imported using the pipeline command. More information is available from [22]. An example command to import a CityGML file is provided in Listing 5.3.



```
(base)                                    /Desktop/3dcitydb-5-devel/postgresql/ShellScripts/Unix"
(base)                         Unix % sh CREATE_DB.sh
 _______   ___ _ _         ___  ___
|__ /   \ / __(_) |_ _  _|   \| _ )
 |_ \ |) | (__| |  _| || | |) | _ \
|___/___/ \___|_|\__|\_, |___/|___/
                     |__/

3D City Database - The Open Source CityGML Database

######################################################################################

Welcome to the 3DCityDB Setup Script. This script will guide you through the process
of setting up a 3DCityDB instance. Please follow the instructions of the script.
Enter the required parameters when prompted and press ENTER to confirm.
Just press ENTER to use the default values.

Documentation and help:
   3DCityDB website:    https://www.3dcitydb.org
   3DCityDB on GitHub:  https://github.com/3dcitydb

Having problems or need support?
   Please file an issue here:
   https://github.com/3dcitydb/3dcitydb/issues

######################################################################################

Please enter a valid SRID (e.g., EPSG code of the CRS to be used).
(SRID must be an integer greater than zero): 28992

Please enter the EPSG code of the height system (use 0 if unknown or '28992' is already 3D).
(default HEIGHT_EPSG=0): 5109

Please enter the corresponding gml:srsName to be used in GML exports.
(default GMLSRSNAME=urn:ogc:def:crs,crs:EPSG::28992,crs:EPSG::5109):

Connecting to "bstsai@localhost:5433/citydb_v5" ...
Password for user bstsai:
SET
 postgis_version
-----------------
 3.3.1

CREATE SCHEMA
  current_path
-----------------
 "$user", public

SET
```

```
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
ALTER DATABASE

3DCityDB creation complete!

Checking spatial reference system ...
 check_srid
------------
 SRID ok

Setting spatial reference system of 3DCityDB instance ...
 change_schema_srid
--------------------

Done

Press ENTER to quit.
```

Figure 5.2.: Create 3DCityDB v.5.0 schema

```
sh citydb import citygml "{directory of CityGML data}"
    --db-host localhost
    --db-port {port number}
    --db-name {database}
    --db-username {user name}
    --db-password {password}
    --db-schema citydb
```

Listing 5.3: Example command to import CityGML file using 3DCityDB v.5.0 tool

5.1.2. QGIS Package Structure and Setup

With a successful 3DCityDB schema setup, users can then install the qgis_pkg developed by this research to create GIS layers from the datasets stored within the 3DCityDB v.5.0 schemas.

The qgis_pkg package consists of seven SQL files, which can only be installed by executing them via pgAdmin4. The main PL/pgSQL functions includes:

- **010_function**: Contains fundamental functions to create the qgis_pkg schema, which holds all other functions used by the package, such as the upsert_extents function for extent selection. It also allows users to create a user schema for storing generated views or materialized views.
- **020_tables**: Establishes the necessary tables to store metadata of the extents, feature geometry, feature attributes, and lookup information.
- **030_meta_geometry**: Contains the update_feature_geometry_metadata main function and other sub-functions for users to check the geometries existence of space, boundary, relief, relief component and address features.

- **040_meta_attribute**: Contains the `update_feature_attribute_metadata` main function and other sub-functions for users to check the existence of geometries related to space, boundary, relief, relief components, and address features.
- **050_meta_layer**: Contains the `drop_all_layer` main function and other side functions to drop views or materialized views of the generated layers.
- **060_geometry**: Contains the `create_geometry_view` main function and other side functions to create or drop views or materialized views of space, boundary, relief, relief components, and address features.
- **070_attribute**: Contains the `create_attribute_view` main function and other side functions to query, flatten, and create feature attribute views or materialized views.
- **080_layers**: Contains the `create_layer` main function and other side functions to create GIS layers using the approaches mentioned in Section 4.4 and store them in views or materialized views.

Users can call the `create_qgis_usr_schema` function (Listing 5.4) after successfully installing the QGIS package to create their 3DCityDB schema for storing the generated views or GIS layers. The EXTENTS and LAYER_METADATA tables are instantiated by duplicating the template tables stored within the `qgis_pkg` schema.

For identification purposes, starting from this chapter, `usr_schema` will refer to the 3DCityDB v.5.0 schemas created by users, whereas `cdb_schema` will denote the 3DCityDB v.5.0 schemas that store the target CityGML datasets.

```sql
-- (input: user_name)
SELECT * FROM qgis_pkg.create_qgis_usr_schema('bstsai');
```

Listing 5.4: Calling the `create_qgis_usr_schema` function

After creating the user schemas, users are requested to specify the bounding box geometry for the operation. There are three fixed types for the bounding box: "db_schema," "m_view," and "qgis," with "db_schema" as the default type. The user-specified extents are classified as "m_view" type, and the "qgis" type is reserved for future development of the plug-in to store the extents provided from QGIS GUIs. Listing 5.5 demonstrates two modes to call the upsert_extents function. Users can either provide the usr_schema and cdb_schema for specifying the extents, or they can specify the bounding box geometry directly.

In the first mode, a full cdb_schema-wise scan is performed to calculate the bounding box geometry of the target dataset and populate it into the EXTENTS table labelled as "db_schema" type. In the second mode, users can specify the bounding box geometry using the `ST_MakeEnvelope` function supported in the PostgreSQL, which takes the four vertex coordinates of the extent regarding the local EPSG code. With the `usr_schema` created and the extent specified, users can then proceed to check feature geometries and attributes for view creation.

```sql
-- (inputs: usr_schema, cdb_schema, cdb_bbox_type, cdb_envelope)
SELECT * FROM qgis_pkg.upsert_extents('qgis_bstsai', 'citydb'); -- Default
SELECT * FROM qgis_pkg.upsert_extents('qgis_bstsai', 'rh_v5', 'm_view',
    ST_MakeEnvelope(232320, 480620, 232615, 481025, 28992)); -- user-specified
```

Listing 5.5: Calling the `upsert_extents` function

5.2. Test Datasets

The datasets used to test the PL/pgSQL functions developed in this research are listed in Table 5.1. The Rijsen-Holten and Vienna datasets are primarily used to evaluate the query performance for collecting and generating views or materialized views of feature geometry and attributes, as they include commonly used features like buildings, trees and terrain. Due to the large size of the Vienna dataset, it is specifically used to test query time efficiency in extreme cases.

Due to the swapped xy-coordinates in the Tokyo CityGML files, additional processing is required. This is supported by the "3D City Database Importer/Exporter," a tool of the 3DCityDB v.4.x, which is used to swap the xy-coordinates and export the modified data. The modified file is then imported by the command line tool of the 3DCityDB v.5.0 to correctly display the data in QGIS. For testing purposes, only a single tile of the Tokyo CityGML dataset is imported into the 3DCityDB v.5.0, particularly to verify feature attributes that are represented in other characters, such as the Kanji in Japanese, can be used as the column headers in the flattened attribute table.

The FZK-Haus and the transportation datasets are used to test the layer generation of CityGML v.3.0 data. They contain features such as building construction elements, stories, and traffic spaces, which are newly introduced in the CityGML v.3.0 standards.

Dataset [source]	Feature Number	Schema Size	CityGML Modules	LoDs	CityGML Version
Alderaan [1]	273	4.3 MB	Building, Vegetation, Relief	0,1,2	2.0
Railway [1]	235	3.0 MB	Bridge, Building, CityFurniture, Generics, Relief, Transportation, Tunnel, Vegetation, WaterBody	3	2.0
Tokyo [38]	39,865	84.0 MB	Building	0,1,2	2.0
Rijsen-Holten [37, 39]	827,105	3.3 GB	Building, Vegetation, Relief	2	2.0
Vienna [40, 41]	7,512,786	38.0 GB	Building, Vegetation, Relief, LandUse, Generics	1,2	2.0
FZK-Haus [42]	85	3.7 MB	Building, Construction	2	3.0
Munich Transportation [43]	295	1.4 MB	Transportation	2	3.0
New York City Transportation [43]	7,179	19 MB	Transportation	2	3.0

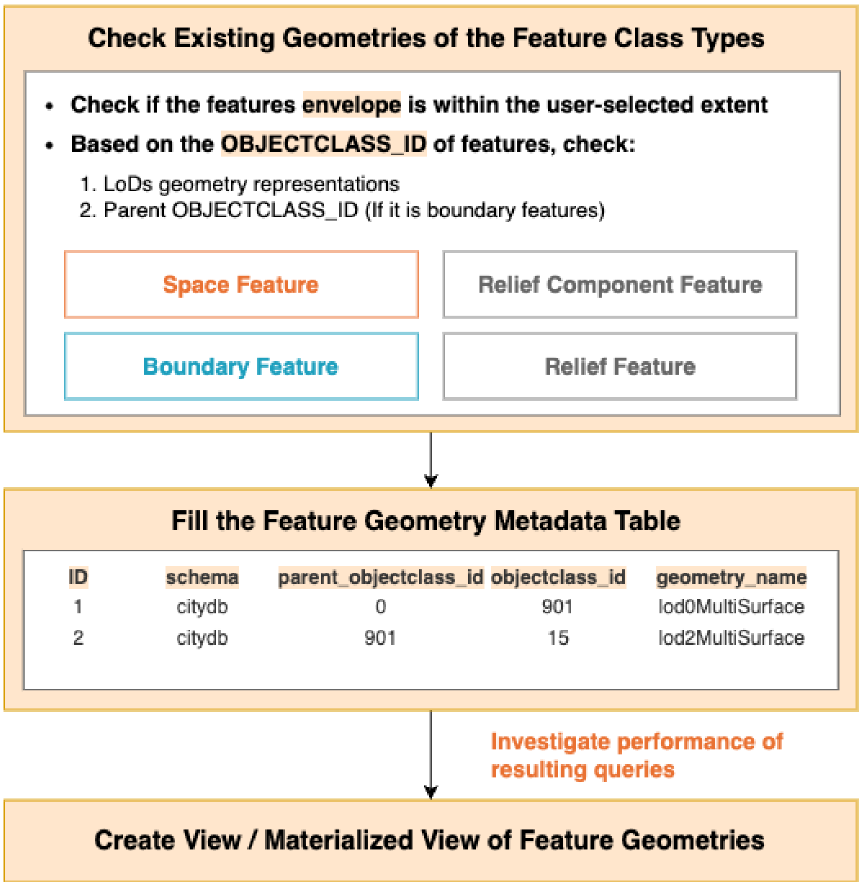
[1] The Alderaan dataset is an artificial CityGML dataset used in the course GEO5014 taught at TU Delft (https://3d.bk.tudelft.nl/education/#courses).

Table 5.1.: List of test datasets and their main properties

5.3. Feature Geometry Metadata Check and View Creation

The overall pipeline of checking existing feature geometries and creating feature geometry views from the data stored within 3DCityDB v.5.0 is shown in Figure 5.3. All features within the user-specified extent can be classified into four categories: Space, Boundary, Relief and Relief Component features. The existing feature geometries in the target database schema can be obtained using a combination of the feature `objectclass_id` and its LoDs geometry property, which is stored in the `name` column of the target schema PROPERTY table. For boundary features, the parent `objectclass_id` should also be checked. The envelope geometries stored in the FEATURE table are needed to reference relief features.

Section 5.3.1 introduces the design of the FEATURE_GEOMETRY_METADATA table, giving information about the columns and their purposes. Section 5.3.2 elaborates on the structure of the PL/pgSQL functions used to perform feature geometry metadata check. Section 5.3.3 explains the PL/pgSQL functions that dynamically generate the queries for creating feature geometry views and demonstrate the result, followed by the summary of this section described in Section 5.3.4.


Check Existing Geometries of the Feature Class Types
• Check if the features envelope is within the user-selected extent
• Based on the OBJECTCLASS_ID of features, check:
1. LoDs geometry representations
2. Parent OBJECTCLASS_ID (If it is boundary features)
Space Feature
Relief Component Feature
Boundary Feature
Relief Feature
Fill the Feature Geometry Metadata Table
Investigate performance of resulting queries
Create View / Materialized View of Feature Geometries

ID	schema	parent_objectclass_id	objectclass_id	geometry_name
1	citydb	0	901	lod0MultiSurface
2	citydb	901	15	lod2MultiSurface

Figure 5.3.: Checking feature geometry metadata pipleline

5.3.1. Geometry Metadata Table Design

As indicated in Section 4.2.1, the FEATURE_GEOMETRY_METADATA table must include at least the following four columns: schema name (`cdb_schema`), parent `objectclass_id`, `objectclass_id`, and geometry name. The distinct combinations of these four keys are used as a constraint in the FEATURE_GEOMETRY_METADATA table to guarantee the uniqueness of each row.

Table 5.2 shows the design of the FEATURE_GEOMETRY_METADATA table. In addition to the four key columns (`schema_name`, `parent_objectclas_id`, `objectclass_id`, and `geometry_name`), the table also includes several other columns. These include `datatype_id`, which indicates whether the geometry type is represented by normal geometries or implicit geometries, and `postgis_geom_type`, which is used for type casting when querying the feature geometry.

The columns after `last_modification_date` are intended for storing information related to geometry views or materialized views. These columns are populated when the corresponding views or materialized views are created, and the values are removed when users drop the views. The view information helps manage the created views, enabling operations such as dropping views or refreshing materialized views. Additionally, `mv_creation_time` and `mv_refresh_time` record the times taken to create or refresh geometry materialized views, respectively, as these operations typically require longer processing times.

Column Name	PostgreSQL Type	Values
id	bigint[PK]	Primary key
cdb_schema	varchar	Target database schema
bbox_type	varchar	Bounding box type for extent selection
parent_objectclass_id	integer	Parent class ID of boundary features. 0 as NULL
parent_classname	varchar	Name of the parent class
objectclass_id	integer	class ID of the features
classname	varchar	Name of the class
datatype_id	integer	ID indicating the property data type, e.g. 11 for geometry property and 16 for implicit geometry property
geometry_name	text	LoD and geometry type, e.g. lod1Solid, lod2MultiSuface, etc.
lod	text	LoD value
geometry_type	text	Geometry type, e.g. Solid, MultiSurface, etc. For naming the geometry views
postgis_geom_type	text	PostGIS geometry type, e.g. PolyhedraSurface for Solid and Multipolygonz for MultiSurface, etc.
last_modification_date	timestamp(3)	Last update time of the record
view_name	varchar	Geometry view name
is_matview	boolean	Identifier of materialized view creation
mview_name	varchar	Geomery materialized view name
mv_creation_time	time(3)	Materialized view creation time
mv_refresh_time	time(3)	Materialized view refresh time
mv_last_update_time	timestamp(3)	Last update time of the materialized view

Table 5.2.: FEATURE_GEOMETRY_METADATA table design

5.3.2. Geometry Metadata Check

The PL/pgSQL functions used to check the metadata of feature geometries are stored in the ”030_meta_geometry” SQL file. The pipeline of the `update_feature_geometry_metadata` function is explained in Algorithm 5.1. The function takes three inputs: `usr_schema`, `cdb_schema` and `cdb_bbox_type`. Users can specify the bounding box type for extent selection. The function populates the FEATURE_GEOMETRY_METADATA table with the check results.

Algorithm 5.1: `update_feature_geometry_metadata`

```
Input: usr_schema, cdb_schema, cdb_bbox_type
Output: void: Populate the FEATURE_GEOMETRY_METADATA table from the
        result of querying existing feature geometries
1  Delete existing records of the specified cdb_schema ;
2  Let $\mathbb{O} \leftarrow$ Array of existing objectclass_ids within the extent ;
3  Let $\mathbb{B} \leftarrow$ Array of Boundary objectclass_ids ;
4  Let $\mathbb{R} \leftarrow$ Relief feature objectclass_id ;
5  for each $o \in \mathbb{O}$ do
6      if $o = \mathbb{R}$ then
7          Call check_relief_feature function with $o$;
8      else if $o \in \mathbb{B}$ then
9          Call check_boundary_feature function with $o$;
10     else
11         Call check_space_feature function with $o$;
```

The check feature geometry metadata process starts with deleting the existing records in the FEATURE_GEOMETRY_METADATA table regarding the given `cdb_schema`, ensuring that the available feature geometries in the specified `cdb_schema` are up-to-date.

For checking the boundary features, an array of all boundary class names is declared in the `update_feature_geometry_metadata` function, which includes the following classes:

- **Core module**: ”ClosureSurface”.
- **Construcntion module**: ”WallSurface”, ”GroundSurface”, ”InteriorWallSurface”, ”RoofSurface”, ”FloorSurface”, ”OuterFloorSurface”, ”CeilingSurface”, ”OuterCeilingSurface” (eight classes in total). Note that ”DoorSurface” and ”WindowSurface” are temporarily classified as space features due to the absence of their parent ”Window” and ”Door” classes, which could have occurred when encoding CityGML v.2.0 dataset using the current 3DCityDB v.5.0 tool.
- **Transportation module**: ”TrafficArea” and ”AuxiliaryTrafficArea”.
- **WaterBody module**: ”WaterSurface”, ”WaterGroundSurface”.

The above thirteen boundary class names array is then converted to an array of the corresponding `objectclass_ids` by looking up the `qgis_pkg` CLASSNAME_LOOKUP table. The same look-up applies to the ”ReliefFeature” class.

Starting from this chapter, the prefixes `qi_` and `ql_` used before parameters denote the `quote_identifier` and `quote_literal` functions in PostgreSQL, respectively. These functions dynamically insert input arguments into the query templates for generating SQL statements.

Listing 5.6 shows the query template for checking the existing `objectclass_ids` from the `cdb_schema` within the specified extent. The query returns distinct `objectclass_ids` as an array while excluding classes from the module “Dynamizer”, “Appearance”, “CityObjectGroup” and “Version”, as they are neither related to feature geometries nor are included for layer creation by the current plug-in.

The sql_where query text shown in Listing 5.6 is added optionally in the check geometry metadata query to perform the extent selection. The geometry of the extent is stored using the Well-Known Binary (WKB) representation, which is referenced from the EXTENTS table under the usr_schema. If the extent selection is enabled, the corresponding geometry will then be extracted from the EXTENTS table and passed to the cdb_envelope variable declared in the update_feature_geometry_metadata function. The bounding box coordinates of the extent will then be computed to make the envelope, and a spatial selection will be performed to test the intersection of the user-defined bounding box and each feature envelope stored within the cdb_schema FEATURE table.

```sql
SELECT ARRAY(
    SELECT DISTINCT objectclass_id
    FROM ',qi_cdb_schema,'.feature AS f
      INNER JOIN ',qi_cdb_schema,'.objectclass AS o ON f.objectclass_id = o.id
      INNER JOIN ',qi_cdb_schema,'.namespace AS n ON o.namespace_id = n.id
    WHERE n.alias NOT IN ("dyn", "app", "grp", "vers")', sql_where,')
        AS oc_ids;
```

Listing 5.6: Query template for checking existing objectclass_ids within the extent

```sql
sql_where := concat(' AND ST_MakeEnvelope(
    ',ST_XMin(cdb_envelope),',',ST_YMin(cdb_envelope),',
    ',ST_XMax(cdb_envelope),',',ST_YMax(cdb_envelope),',',srid,') && f.
    envelope ');
```

Listing 5.7: Query template for extent selection

The update_feature_geometry_metadata function traverses through the array of distinct objectclass_ids to call corresponding functions for populating the geometry metadata table. The ”ReliefFeatures” class is checked first because it has special geometry representations (see Section 4.2.1). If relief features are found, the function named check_relief_feature is invoked to populate the geometry metadata table with the available relief feature LoDs and geometry representations, as shown in Listing 5.8.

```sql
SELECT DISTINCT
    f.objectclass_id,
    p.datatype_id,
    p.name,
    p.val_int
FROM ',qi_cdb_schema,'.feature AS f
    INNER JOIN ',qi_cdb_schema,'.property AS p ON f.id = p.feature_id
        AND f.objectclass_id = 500 -- ReliefFeature class
WHERE p.name = 'lod' ', sql_where, ';
```

Listing 5.8: Query excerpt for checking relief feature

The boundary classes are checked after the ”ReliefFeature” class because they are associated with their parent classes. Listing 5.9 presents a query excerpt to check the LoDs and geometry representations of boundary classes in the function named check_boundary_feature. The boundary classes' geometry metadata check is achieved by repetitively cross-referencing the cdb_schema FEATURE and PROPERTY tables. The first join identifies the "boundary" property of the space features, and the boundary class is specified in the second join. The datatype_id is set to 11 here since boundary features can only have a geometry property but no implicit geometry property (see Section 4.2.1). The objectclass_id of "RoofSurface" (712) is specified here as an example; the boundary objectclass_id is dynamically specified based on different boundary classes, returning the possible geometry metadata in combinations of space-boundary classes like Building-RoofSurface or Tunnel-RoofSurface.

```sql
SELECT DISTINCT
    f1.objectclass_id AS parent_obejectclass_id,
    f2.objectclass_id,
    p2.name,
    p2.datatype_id,
    p2.name,
    p2.val_lod
FROM ',qi_cdb_schema,'.feature AS f1
    INNER JOIN ',qi_cdb_schema,'.property AS p1 ON f1.id = p1.feature_id
        AND p1.name = 'boundary'
    INNER JOIN ',qi_cdb_schema,'.feature AS f2 ON f2.id = p1.val_feature_id
        AND f2.objectclass_id = 712 ',sql_where,' --RoofSurface class
    INNER JOIN ',qi_cdb_schema,'.property AS p2 ON f2.id = p2.feature_id
        AND p2.datatype_id = 11; -- geometry property
```

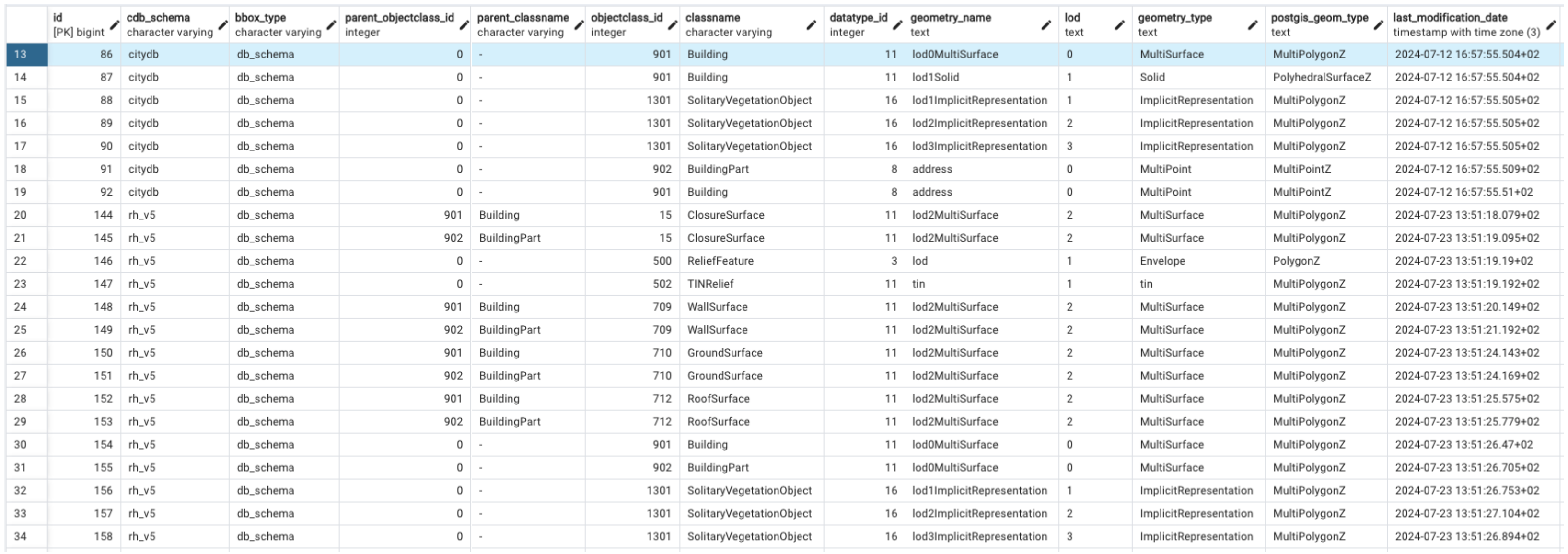
Listing 5.9: Query excerpt for checking boundary features

Finally, the remaining features are the space and relief component features, which share the same query concept for checking their LoDs and geometry representations. Listing 5.10 provides a query excerpt from the function named check_space_feature. The space and relief component geometry metadata check is achieved by cross-referencing the cdb_schema FEATURE and PROPERTY tables once. Since space features can have both geometry and implicit geometry properties (see Section 4.2.1), the datatype_id is set to include 11 and 16, representing these two types. The Building class is used as an example specified by the objectclass_id 901, which can be dynamically specified based on different space or relief component feature classes, providing the geometry metadata for classes such as "Building", "SolitaryVegetationObject", and "TINRelief".

```sql
SELECT DISTINCT
    f.objectclass_id,
    p.name,
    p.datatype_id,
    p.name,
    p.val_lod
FROM ',qi_cdb_schema,'.feature AS f
    INNER JOIN ',qi_cdb_schema,'.property AS p ON f.id = p.feature_id
        AND f.objectclass_id = 901 ',sql_where,' --Building class
WHERE p.datatype_id IN (11,16) AND p.val_lod IS NOT NULL;
```

Listing 5.10: Query excerpt for checking space and relief component features

The query excerpts for checking the feature geometry metadata can be extended to process and return additional information, such as geometry_type and postgis_geometry_type, as introduced in Table 5.2. An example of the populated FEATURE_GEOMETRY_METADATA table is shown in Figure 5.4.



	id [PK] bigint	cdb_schema character varying	bbox_type character varying	parent_objectclass_id integer	parent_classname character varying	objectclass_id integer	classname character varying	datatype_id integer	geometry_name text	lod text	geometry_type text	postgis_geom_type text	last_modification_date timestamp with time zone (3)
13	86	citydb	db_schema	0	-	901	Building	11	lod0MultiSurface	0	MultiSurface	MultiPolygonZ	2024-07-12 16:57:55.504+02
14	87	citydb	db_schema	0	-	901	Building	11	lod1Solid	1	Solid	PolyhedralSurfaceZ	2024-07-12 16:57:55.504+02
15	88	citydb	db_schema	0	-	1301	SolitaryVegetationObject	16	lod1ImplicitRepresentation	1	ImplicitRepresentation	MultiPolygonZ	2024-07-12 16:57:55.505+02
16	89	citydb	db_schema	0	-	1301	SolitaryVegetationObject	16	lod2ImplicitRepresentation	2	ImplicitRepresentation	MultiPolygonZ	2024-07-12 16:57:55.505+02
17	90	citydb	db_schema	0	-	1301	SolitaryVegetationObject	16	lod3ImplicitRepresentation	3	ImplicitRepresentation	MultiPolygonZ	2024-07-12 16:57:55.505+02
18	91	citydb	db_schema	0	-	902	BuildingPart	8	address	0	MultiPoint	MultiPointZ	2024-07-12 16:57:55.509+02
19	92	citydb	db_schema	0	-	901	Building	8	address	0	MultiPoint	MultiPointZ	2024-07-12 16:57:55.51+02
20	144	rh_v5	db_schema	901	Building	15	ClosureSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:18.079+02
21	145	rh_v5	db_schema	902	BuildingPart	15	ClosureSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:19.095+02
22	146	rh_v5	db_schema	0	-	500	ReliefFeature	3	lod	1	Envelope	PolygonZ	2024-07-23 13:51:19.19+02
23	147	rh_v5	db_schema	0	-	502	TINRelief	11	tin	1	tin	MultiPolygonZ	2024-07-23 13:51:19.192+02
24	148	rh_v5	db_schema	901	Building	709	WallSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:20.149+02
25	149	rh_v5	db_schema	902	BuildingPart	709	WallSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:21.192+02
26	150	rh_v5	db_schema	901	Building	710	GroundSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:24.143+02
27	151	rh_v5	db_schema	902	BuildingPart	710	GroundSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:24.169+02
28	152	rh_v5	db_schema	901	Building	712	RoofSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:25.575+02
29	153	rh_v5	db_schema	902	BuildingPart	712	RoofSurface	11	lod2MultiSurface	2	MultiSurface	MultiPolygonZ	2024-07-23 13:51:25.779+02
30	154	rh_v5	db_schema	0	-	901	Building	11	lod0MultiSurface	0	MultiSurface	MultiPolygonZ	2024-07-23 13:51:26.47+02
31	155	rh_v5	db_schema	0	-	902	BuildingPart	11	lod0MultiSurface	0	MultiSurface	MultiPolygonZ	2024-07-23 13:51:26.705+02
32	156	rh_v5	db_schema	0	-	1301	SolitaryVegetationObject	16	lod1ImplicitRepresentation	1	ImplicitRepresentation	MultiPolygonZ	2024-07-23 13:51:26.753+02
33	157	rh_v5	db_schema	0	-	1301	SolitaryVegetationObject	16	lod2ImplicitRepresentation	2	ImplicitRepresentation	MultiPolygonZ	2024-07-23 13:51:27.104+02
34	158	rh_v5	db_schema	0	-	1301	SolitaryVegetationObject	16	lod3ImplicitRepresentation	3	ImplicitRepresentation	MultiPolygonZ	2024-07-23 13:51:26.894+02

Figure 5.4.: Populated FEATURE_GEOMETRY_METADATA table example

5.3.3. Geometry View Creation

After performing the geometry metadata scan, users can check the availability of feature geometry representations and LoDs in the FEATURE_GEOMETRY_METADATA table. The main function used to create feature geometry views is named `create_geometry_view`, which is wrapped with other additional functions in the "050_geometry" SQL file.

Algorithm 5.2 shows the pipeline of the `create_geometry_view` function. The function allows users to create views or materialized views of the selected feature geometry representation and LoD within an extent according to the input geometry metadata parameters. Users can specify which view types by the boolean parameter, `is_matview`, and the query texts of view headers are generated by `generate_sql_view_header` or `generate_sql_matview_header` correspondingly. Additionally, the `generate_sql_matview_footer` function is called only for the materialized views to add indices on their columns to enhance query time efficiency. These functions are stored in the "010_functions" SQL file, and the view headers and footers are then added to the collect feature geometry queries for geometry view creation.

The character varying parameter, `cdb_bbox_type`, can be specified for extent selection. The `is_matview` parameter is set to false by default for views while the `cdb_bbox_type` is set to "db_schema" for full dataset extent by default.

Algorithm 5.2: `create_geometry_view`

```
Input: usr_schema, cdb_schema, parent_objectclass_id ($po$), objectclass_id ($o$),
       datatype_id ($o_d$), geometry_name, lod, geometry_type, postgis_geom_type,
       is_matview, cdb_bbox_type
Output: void: Create feature geometry views or materialized views and store in
        usr_schema
1  Let $\mathbb{G} \leftarrow$ datatype_id of the geometry property ;
2  Let $\mathbb{I} \leftarrow$ datatype_id of the implicit geometry property ;
3  Let $\mathbb{R} \leftarrow$ Relief feature objectclass_id ;
4  Let $\mathbb{C} \leftarrow$ Array of Relief Component feature objectclass_ids ;
5  Let $sql\_header \leftarrow$ generated based on the is_matview input;
6  Let $sql\_footer \leftarrow$ generated only when the is_matview is TRUE;
7  if $o = \mathbb{R}$ then
8      $sql\_geom \leftarrow$ Call collect_geometry_relief_feature function with $o$;
9  else if $o \in \mathbb{C}$ then
10     $sql\_geom \leftarrow$ Call collect_geometry_relief_component function with $o$;
11 else
12     if $po = 0$ then
13         if $o_d \in \{\mathbb{G}, \mathbb{I}\}$ then
14             $sql\_geom \leftarrow$ Call collect_geometry_space_feature function with $o$;
15     else
16         if $o_d = \mathbb{G}$ then
17             $sql\_geom \leftarrow$ Call collect_geometry_boundary_feature function with $po$
               and $o$;
18 execute $(sql\_header \oplus sql\_geom \oplus sql\_footer)$;
```

The function creates geometry views based on the four feature types: space, boundary, relief, and relief component features. The `objectclass_ids` differentiate the relief and relief component features from space and boundary features, while the `parent_objectclass_ids` can distinguish space and boundary features, as space features have 0 by default for that. The `datatype_ids` for geometry and implicit geometry property are referenced to look up the geometry root `ids` for joining the `cdb_schema` GEOMETRY_DATA table.

Listing 5.11 and Listing 5.12 present the query templates for adding the view and materialized view headers and materialized view footers. The `qgis_pkg` CLASSNAME_LOOKUP table is referenced to get the class name alias for view name creation, such as "bdg" stands for Building class. The naming convention of the view names concatenates the `cdb_schema` name, class name alias, LoD value, and the geometry representation using underscores. For example, if users choose to create a geometry view of buildings in LoD1 Solid from the "citydb" schema, the view name will be: "citydb_bdg_lod1_Solid". In the case of a materialized view, the "_g_" label is added before the view name for identification. The geometry metadata, which are the input parameters of the `create_geometry_view` function, are used to create the view name dynamically within the function, and the view headers and footers are generated correspondingly.

```sql
-- view header
DROP VIEW IF EXISTS ',qi_usr_schema,'.',qi_view_name,' CASCADE;
CREATE OR REPLACE VIEW ',qi_usr_schema,'.',qi_view_name,' AS
-- materialized view header
DROP MATERIALIZED VIEW IF EXISTS ',qi_usr_schema,'.',qi_view_name,' CASCADE;
CREATE MATERIALIZED VIEW ',qi_usr_schema,'.',qi_view_name,' AS
```

Listing 5.11: Query template for view and materialized view headers

```sql
-- materialized view footer
CREATE INDEX ',f_idx_name,' -- index name of feature ID
    ON ',qi_usr_schema,'.',qi_view_name,' (f_id);
CREATE INDEX ',fo_idx_name,' -- index name of feature objectid
    ON ',qi_usr_schema,'.',qi_view_name,' (f_object_id);
CREATE INDEX ',geom_spx_name,' -- index name of geometry
    ON ',qi_usr_schema,'.',qi_view_name,' USING gist (geom);
ALTER TABLE ',qi_usr_schema,'.',qi_view_name,' OWNER TO ',qi_usr_name,';
```

Listing 5.12: Query template for materialized view footers

The following sub-sections present the query templates to collect geometries of relief, relief component, space, and boundary features. The geometry metadata gathered from the FEATURE_GEOMETRY_METADATA table is passed dynamically to these templates by the `create_geometry_view` function to collect the feature geometries. The `sql_where` query can also be added in the query to perform extent selection.

5.3.3.1. Relief and Relief Component features Geometry Query

Listing 5.13 and Listing 5.14 show the query templates to collect geometries of relief and relief component features. The feature envelopes stored within the `cdb_schema` FEATURE table represent the geometries of relief features, and there are no geometry types for them, as they are the bounding boxes of the relief component features. The LoDs values are stored in the `val_int` column within the `cdb_schema` PROPERTY table.

```
SELECT
    f.id::bigint     AS f_id,
    f.objectid::text AS f_object_id,
    f.envelope::geometry(',ql_post_geom_type,',',srid,') AS geom
FROM ',qi_cdb_schema,'.property p
    INNER JOIN ',qi_cdb_schema,'.feature f ON (p.feature_id = f.id
        AND f.objectclass_id = ', oc_id,'  -- Target feature class
        AND p.val_int = ',ql_lod,' -- LoD value
        AND p.name = 'lod'',sql_where,');
```

Listing 5.13: Query template for collecting geometries of relief features

For the relief component features, the geometry types are specified in the `name` column of the `cdb_schema` PROPERTY table, with their LoDs values stored in the val_lod column. The geometry root IDs are stored in the `val_geometry_id` column, which is joined with the ids within the `cdb_schema` GEOMETRY_DATA table to access the geometries.

```
SELECT
    f.id::bigint        AS f_id,
    f.objectid::text    AS f_object_id,
    g.geometry::geometry(',ql_post_geom_type,',',srid,') AS geom
FROM ',qi_cdb_schema,'.property AS p
    INNER JOIN ',qi_cdb_schema,'.feature AS f ON (p.feature_id = f.id
        AND f.objectclass_id = ', oc_id,' -- Target feature class
        AND p.name = ',ql_geom_name,' -- geometry type
        AND p.val_lod = ',ql_lod,'',sql_where,') -- LoD value
    INNER JOIN ',qi_cdb_schema,'.geometry_data AS g
        ON (p.val_geometry_id = g.id AND g.geometry IS NOT NULL);
```

Listing 5.14: Query template for collecting geometries of relief component features

5.3.3.2. Space feature Geometry Query

Listing 5.15 and Listing 5.16 give the query templates to collect geometries and implicit geometries of space features. The space feature geometries query is similar to that of the relief component features. The LoDs and geometry representations of space features geometries are stored together in the `name` column of the `cdb_schema` PROPERTY table, while the geometry root `ids` are stored in the `val_geometry_id` column, associating to the corresponding space feature geometries in the `cdb_schema` GEOMETRY_DATA table.

```
SELECT
    f.id::bigint        AS f_id,
    f.objectid::text    AS f_object_id,
    g.geometry::geometry(',ql_post_geom_type,',',srid,') AS geom
FROM ',qi_cdb_schema,'.property AS p
    INNER JOIN ',qi_cdb_schema,'.feature AS f ON (p.feature_id = f.id
        AND f.objectclass_id = ',oc_id,' -- Target feature class
        AND p.name = ',ql_geom_name,' -- LoD and geometry type
        AND p.val_geometry_id IS NOT NULL ',sql_where,')
    INNER JOIN ',qi_cdb_schema,'.geometry_data AS g
        ON (p.val_geometry_id = g.id AND g.geometry IS NOT NULL);
```

Listing 5.15: Query template for collecting geometries of space features

Space features can have implicit representations according to the geometry and LoD concepts of CityGML v.3.0 (see Figure 2.6 and Table 4.1). A common example of such a space feature class is the "SolitaryVegetationObject" class for trees. The implicit representation of the space features is achieved by scaling the template geometry and moving it to the reference points repetitively. The LoDs and geometry representations of space feature implicit representations are specified in the `name` column, while the reference point geometry and scaling coefficients array are stored in the `val_implicitgeom_refpoint` and `val_array` columns, respectively, in the `cdb_schema` PROPERTY table.

```sql
SELECT
    f.id::bigint        AS f_id,
    f.objectid::text    AS f_object_id,
    st_setsrid(
            st_translate(
                    st_affine(
                            g.implicit_geometry,
                            (val_array->>0) ::double precision,
                            (val_array->>1) ::double precision,
                            (val_array->>2) ::double precision,
                            (val_array->>4) ::double precision,
                            (val_array->>5) ::double precision,
                            (val_array->>6) ::double precision,
                            (val_array->>8) ::double precision,
                            (val_array->>9) ::double precision,
                            (val_array->>10)::double precision,
                            (val_array->>3) ::double precision,
                            (val_array->>7) ::double precision,
                            (val_array->>11)::double precision
                            ),
                    st_x(p.val_implicitgeom_refpoint),
                    st_y(p.val_implicitgeom_refpoint),
                    st_z(p.val_implicitgeom_refpoint)
                    ),
                    ',srid,'
            )::geometry(',ql_postgis_geom_type,',',srid,') AS geom
FROM ',qi_cdb_schema,'.property p
    INNER JOIN ',qi_cdb_schema,'.feature AS f ON (p.feature_id = f.id
        AND f.objectclass_id = ', oc_id,' -- Target feature class
        AND p.name = ',ql_geom_name,'  -- LoD value and geometry type
        AND p.val_implicitgeom_id IS NOT NULL',sql_where,')
    INNER JOIN ',qi_cdb_schema,'.implicit_geometry AS ig
        ON (p.val_implicitgeom_id = ig.id
        AND p.val_implicitgeom_id IS NOT NULL)
    INNER JOIN ',qi_cdb_schema,'.geometry_data AS g
        ON (ig.relative_geometry_id = g.id
        AND g.implicit_geometry IS NOT NULL);
```

Listing 5.16: Query template for collecting implicit representations of space features

In Listing 5.16, `cdb_schema` FEATURE and PROPERTY tables are first cross-referenced to query the LoD and implicit representation. The implicit geometry roots are referenced by joining the ID in `cdb_schema` IMPLICIT_GEOMETRY table on the `val_implicitgeom_id`, which then gives the `relative_geometry_id` to associate with the template geometry stored within the `cdb_schema` GEOMETRY_DATA table.

The `ST_Affine` and `ST_Translate` functions supported in PostgreSQL are called to perform the scaling and moving of the implicit geometry template. The scaling coefficients array has twelve decimal numbers, representing the transformation matrix [44]. The implicit geometry template is first affined and then translated to the x, y, and z coordinates of the reference point regarding the local SRID [45]. After the scaling and translating, the template geometry is spatially populated to the reference points for the space feature implicit representation.

5.3.3.3. Boundary feature Geometry Query

Listing 5.17 presents the query template for collecting geometries of boundary features. The query involves cross-referencing the `cdb_schema` FEATURE and PROPERTY tables twice to establish the hierarchical relationships between boundary features and their parent space features. Similar to the approach used for checking geometry metadata (see Listing 5.9), the first join identifies the "boundary" property of the space features, specifying the `objectclass_id` of the parent space feature. In the second join, the query specifies the boundary feature class, LoD, geometry representation, and data type, which helps to narrow down the target boundary feature class and retrieve the geometry root `ids`. Finally, the `id` in the `cdb_schema` GEOMETRY_DATA table is joined with `val_geometry_id` to access the boundary feature geometries.

```sql
SELECT
    f1.id::bigint        AS f_id,
    f1.objectid::text    AS f_object_id,
    g.geometry::geometry(',ql_post_geom_type,',',srid,') AS geom
FROM ', qi_cdb_schema,'.feature AS f
    INNER JOIN ',qi_cdb_schema,'.property AS p ON (f.id = p.feature_id
        AND p.name = 'boundary'
        AND f.objectclass_id = ', p_oc_id,') -- Parent feature class
    INNER JOIN ',qi_cdb_schema,'.feature AS f1 ON f1.id = p.val_feature_id
    INNER JOIN ',qi_cdb_schema,'.property AS p1 ON (f1.id = p1.feature_id
        AND f1.objectclass_id = ', oc_id,' -- Target feature class
        AND p1.datatype_id = ',geom_datatype_id,' -- only geometry property
        AND p1.name = ',ql_geom_name,') -- LoD value and geometry type
        ',sql_where,'
    INNER JOIN ',qi_cdb_schema,'.geometry_data AS g
        ON (p1.val_geometry_id = g.id
        AND g.geometry IS NOT NULL);
```

Listing 5.17: Query template for collecting geometries of boundary features

5.3.3.4. Geometry View Results

The `create_geometry_view` function uses the geometry metadata retrieved from the FEATURE_GEOMETRY_METADATA table as inputs (see Algorithm 5.2) to dynamically generate SQL statements. These statements are executed to create feature geometry views or materialized views based on the templates mentioned above. Listing 5.18 demonstrates examples of creating a materialized view of buildings in LoD1 Solid from the default "citydb" schema (Alderaan dataset). The `refresh_geometry_materialized_view` function is called to refresh the generated feature geometry materialized view, which is designated to be executed separately as it could take a long time to refresh in case of large datasets. The created results are stored in the `usr_schema` named "qgis_bstsai", and the view names and other metadata regarding the views are updated to the geometry metadata table.

There are three columns in the feature geometry views: `f_id`, `f_object_id` and the geometries, where the first two columns are stored for feature identification. Users can drag these geometry views in QGIS for visualisation both in 2D and 3D (Figure 5.5 and Figure 5.6). To create geometry views, users can leave the last input, `is_matview`, as it is set to false by default.

```sql
-- Materialized View: buildings in LoD1 Solid
SELECT qgis_pkg.create_geometry_view('qgis_bstsai', 'citydb',
    0, 901, 11, 'lod1Solid', '1', 'Solid', 'PolyhedralSurfaceZ', TRUE);
-- Refresh the generated materialized view
SELECT qgis_pkg.refresh_geometry_materialized_view
    ('qgis_bstsai', 'citydb', 0, 901, 'lod1Solid');
```

Listing 5.18: Calling the `create_geometry_view` function

Additional functions for users to manage the feature geometry views are included in the ”050_geometry” SQL file. For instance, the `create_all_geometry_view_in_schema` function allows users to bulk-create views of available feature geometries within an extent from the selected `cdb_schema`. Listing 5.19 shows two modes for bulk creation of views. Users can specify the target class to create geometry views of that class or use the default inputs to create all available feature geometry views. The `refresh_all_geometry_materialized_view` function is called to refresh bulk-created geometry views correspondingly.

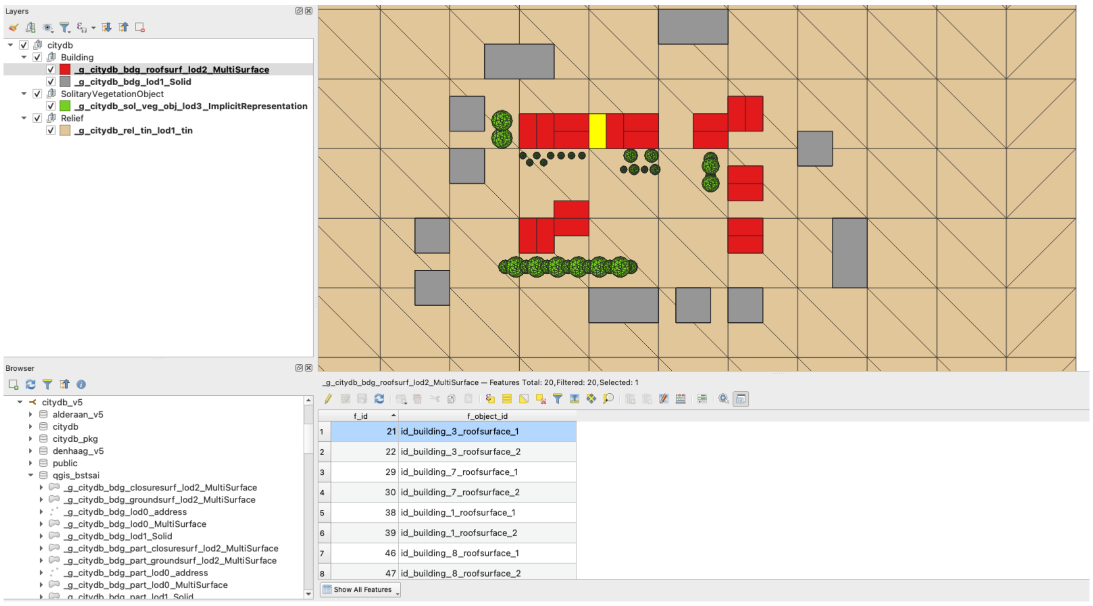
```sql
-- Inputs (default value): usr_schema, cdb_schema, parent_oc_id (NULL), oc_id
    (NULL), is_matview (FALSE), cdb_bbox_type ('db_schema')
-- All classes
SELECT qgis_pkg.create_all_geometry_view_in_schema
    ('qgis_bstsai', 'citydb'); -- views
SELECT qgis_pkg.create_all_geometry_view_in_schema
    ('qgis_bstsai', 'citydb', NULL, NULL, TRUE); -- materialized views
SELECT qgis_pkg.refresh_all_geometry_materialized_view
    ('qgis_bstsai', 'citydb');
-- Building class only
SELECT qgis_pkg.create_all_geometry_view_in_schema
    ('qgis_bstsai', 'citydb', 0, 901); -- views
SELECT qgis_pkg.create_all_geometry_view_in_schema
    ('qgis_bstsai', 'citydb', 0, 901, TRUE); -- materialized views
SELECT qgis_pkg.refresh_all_geometry_materialized_view
    ('qgis_bstsai', 'citydb', 0, 901);
```

Listing 5.19: Calling the `create_all_geometry_view_in_schema` function

Finally, the generated geometry views can be dropped individually or jointly by calling the `drop_all_geometry_views` function. Listing 5.20 shows two modes to drop the generated geometry views. The metadata of the view names and the view creation time are removed from the FEATURE_GEOMETRY_METADATA table.

```sql
-- Inputs (default value): usr_schema, cdb_schema, parent_oc_id (NULL), oc_id
    (NULL)
-- All classes
SELECT qgis_pkg.drop_all_geometry_views('qgis_bstsai', 'citydb');
-- Building class only
SELECT qgis_pkg.drop_all_geometry_views('qgis_bstsai', 'citydb', 0, 901);
```

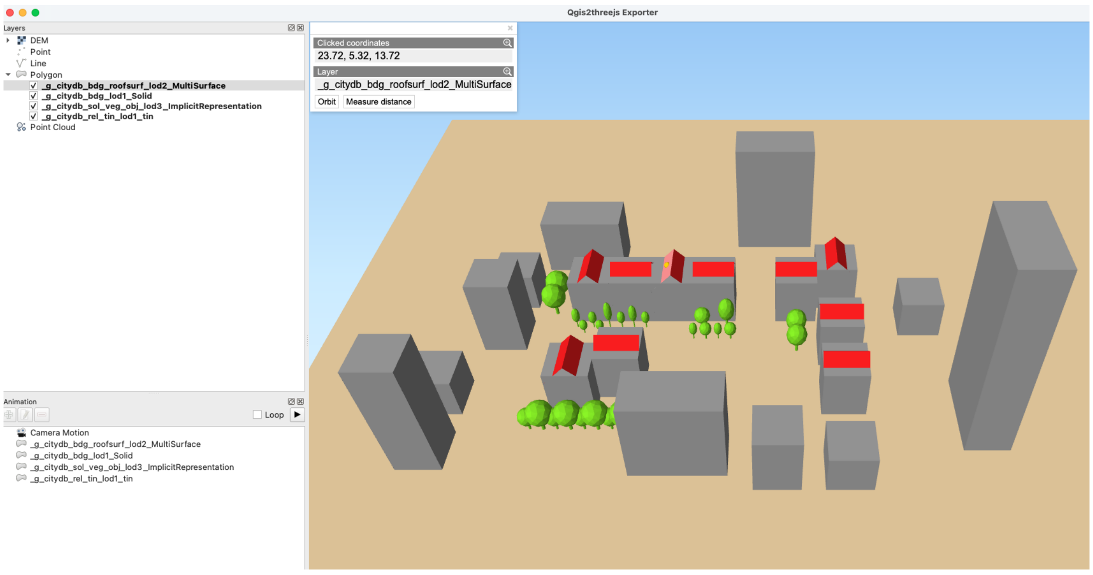
Listing 5.20: Calling the `drop_all_geometry_view` function


Layers
citydb
Building
_g_citydb_bdg_roofsurf_lod2_MultiSurface
_g_citydb_bdg_lod1_Solid
SolitaryVegetationObject
_g_citydb_sol_veg_obj_lod3_ImplicitRepresentation
Relief
_g_citydb_rel_tin_lod1_tin
Browser
citydb_v5
alderaan_v5
citydb
citydb_pkg
denhaag_v5
public
qgis_bstsai
_g_citydb_bdg_closuresurf_lod2_MultiSurface
_g_citydb_bdg_groundsurf_lod2_MultiSurface
_g_citydb_bdg_lod0_address
_g_citydb_bdg_lod0_MultiSurface
_g_citydb_bdg_lod1_Solid
_g_citydb_bdg_part_closuresurf_lod2_MultiSurface
_g_citydb_bdg_part_groundsurf_lod2_MultiSurface
_g_citydb_bdg_part_lod0_address
_g_citydb_bdg_part_lod0_MultiSurface
Show All Features

_g_citydb_bdg_roofsurf_lod2_MultiSurface — Features Total: 20,Filtered: 20,Selected: 1

	f_id	f_object_id
1	21	id_building_3_roofsurface_1
2	22	id_building_3_roofsurface_2
3	29	id_building_7_roofsurface_1
4	30	id_building_7_roofsurface_2
5	38	id_building_1_roofsurface_1
6	39	id_building_1_roofsurface_2
7	46	id_building_8_roofsurface_1
8	47	id_building_8_roofsurface_2

Figure 5.5.: Feature geometry materialized views 2D visualisation in QGIS (Alderaan dataset)


Qgis2threejs Exporter
Layers
DEM
Point
Line
Polygon
_g_citydb_bdg_roofsurf_lod2_MultiSurface
_g_citydb_bdg_lod1_Solid
_g_citydb_sol_veg_obj_lod3_ImplicitRepresentation
_g_citydb_rel_tin_lod1_tin
Point Cloud
Animation
Loop
Camera Motion
_g_citydb_bdg_roofsurf_lod2_MultiSurface
_g_citydb_bdg_lod1_Solid
_g_citydb_sol_veg_obj_lod3_ImplicitRepresentation
_g_citydb_rel_tin_lod1_tin
Orbit
Measure distance

Clicked coordinates
23.72, 5.32, 13.72
Layer
_g_citydb_bdg_roofsurf_lod2_MultiSurface

Figure 5.6.: Feature geometry materialized views 3D visualisation in QGIS (Alderaan dataset)

5.3.4. Lessons Learned from the Feature Geometries

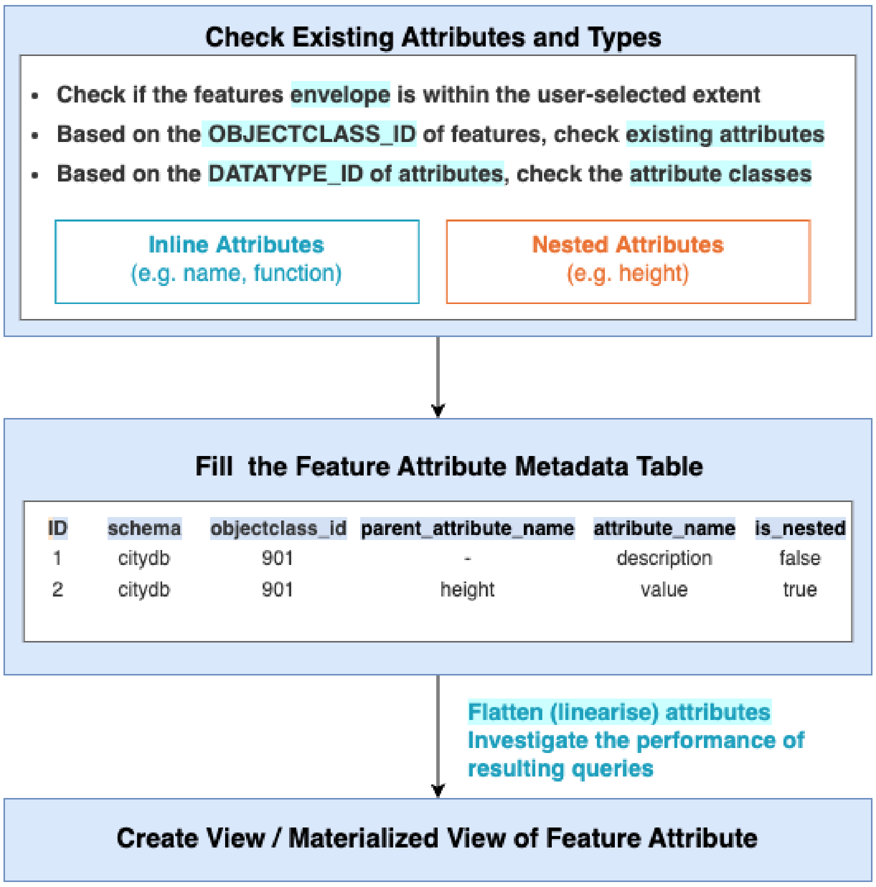
The 3DCityDB v.5.0 encoding simplifies feature geometry view creation as geometry roots are directly associated with geometries, unlike the decomposed storage in 3DCityDB v.4.x. Compared to 3DCityDB v.4.x, it is not necessary to create materialized views for all feature geometry representations (see Section 4.2.2). However, the integrated storage increases the size of the FEATURE and PROPERTY tables, impacting query performance, especially with large datasets. Thus, using materialized views is still preferable for faster queries.

The server-side `qgis_pkg` developed by this research allows users to select specific LoDs and geometry representations for each class in a CityGML module, offering more flexibility than the current plug-in, which creates views for all classes in a module. For example, users can choose to create geometry views of buildings in LoD1 Solid, while the current plug-in creates views for all LoDs and geometry representations of classes like ”Building” and ”BuildingPart” within the ”Building” module. This enhances user interaction with feature geometries from CityGML data stored in 3DCityDB.

5.4. Feature Attribute Metadata Check and View Creation

The overall pipeline of checking existing feature attributes and creating feature attribute views from the data stored within 3DCityDB v.5.0 is shown in Figure 5.7. Feature attributes within the user-specified extent can be classified into two main classes: Inline and Nested attributes. The existing feature attributes in the target database schema can be obtained by querying the distinct properties of each `objectclass_id`. The feature attribute names are indicated by the `name` column of the target schema PROPERTY table, with their values stored across the corresponding columns based on the `datatype_ids`. The Inline-Nested attribute classes can then be determined by the `datatype_ids` and exclude the spatial, address, feature boundary and appearance properties.

Section 5.4.1 introduces the design of the FEATURE_ATTRIBUTE_METADATA table, giving information about the columns and their purposes. Section 5.4.2 elaborates on the structure of the PL/pgSQL functions used to perform feature attribute metadata checks. Section 5.4.3 explains the PL/pgSQL functions to dynamically generate queries for flattening and creating feature attribute views, and Section 5.4.4 provides the section summary.


Check Existing Attributes and Types
• Check if the features envelope is within the user-selected extent
• Based on the OBJECTCLASS_ID of features, check existing attributes
• Based on the DATATYPE_ID of attributes, check the attribute classes
Inline Attributes
(e.g. name, function)
Nested Attributes
(e.g. height)
Fill the Feature Attribute Metadata Table
Flatten (linearise) attributes
Investigate the performance of resulting queries
Create View / Materialized View of Feature Attribute

ID	schema	objectclass_id	parent_attribute_name	attribute_name	is_nested
1	citydb	901	-	description	false
2	citydb	901	height	value	true

Figure 5.7.: Checking feature attribute metadata pipleline

5.4.1. Attribute Metadata Table Design

As indicated in Section 4.3.1, the FEATURE_ATTRIBUTE_METADATA table must include at least the following four columns: schema name (`cdb_schema`), `objectclass_id`, attribute name and attribute data type. The distinct combinations of these four keys are used as a constraint in the FEATURE_ATTRIBUTE_METADATA table to guarantee the uniqueness of each row.

Table 5.3 shows the design of the FEATURE_ATTRIBUTE_METADATA table. In addition to the four key columns: `schema_name`, `objectclass_id`, `attribute_name` and `attribute_datatype`, the table also includes several other columns:

- **is_nested**: Identify the attribute classes. It calls the corresponding PL/pgSQL functions to collect and flatten (linearise) the Inline or Nested attributes.
- **is_multiple** and **max_multiplicity**: Identify whether the multiplicity of an attribute is greater than one and record the number of multiplicity. The multiplicity information is crucial for using the crosstab function and determining the number of value columns in the crosstab `FROM` clause. These two columns are populated when users choose to create views of the attribute.
- **is_multiple_value_columns**, **value_column** and **n_value_columns**: Identify whether the attribute values are stored across multiple value columns in the `cdb_schema` PROPERTY table. The `value_column` stores the name of the attribute value column in an array, while the number of the columns is indicated in the `n_value_columns`. These three columns are populated when users choose to create views of the attribute.
- **ct_type_name**: Stores the composite type names, which are used when the crosstab function is called to flatten the selected attribute that has multiplicity greater than one. The composite type names are included for managing attribute views, which are also populated when users choose to create views of the attribute.

The columns after the `last_modification_date` are intended for storing information related to geometry views or materialized views. These columns are populated when the corresponding views or materialized views are created, and the values are removed when users drop the views. The view information helps manage the created views, enabling operations such as dropping views or refreshing materialized views.

Column Name	PostgreSQL Type	Values
id	bigint[PK]	Primary key
cdb_schema	varchar	Target database schema
bbox_type	varchar	Bounding box type for extent selection
objectclass_id	integer	class ID of the features
classname	varchar	Name of the class
parent_attribute_name	varchar	Nested parent attribute "-" as NULL
parent_attribute_typename	varchar	Nested parent attribute data type "-" as NULL
attri_name	varchar	Inline / Nested child attribute
attribute_typename	varchar	Inline / Nested child attribute data type
is_nested	boolean	Identifier of Nested attribute
is_multiple	boolean	Identifier of multiple entries
max_multiplicity	integer	number of multiplicity
is_multiple_value_columns	boolean	Identifier multiple value columns
ct_type_name	varchar	composite type names
n_value_columns	integer	number of value columns
value_column	text[]	Array of value column names
last_modification_date	timestamp(3)	Last update time of the record
view_name	varchar	Attribute view name
view_creation_date	timestamp(3)	Attribute view creation date
mview_name	varchar	Attribute materialized view name
mv_refresh_date	timestamp(3)	Attribute materialized view refresh date

Table 5.3.: FEATURE_ATTRIBUTE_METADATA table design

5.4.2. Attribute Metadata Check

The PL/pgSQL functions used to check the metadata of feature attributes are stored in the "040_meta_attribute" SQL file. The pipeline of the `update_feature_attribute_metadata` function is explained in Algorithm 5.3. The function takes three inputs: `usr_schema`, `cdb_schema` and cdb_bbox_type. Users can specify the bounding box type for extent selection. The function populates the FEATURE_ATTRIBUTE_METADATA table with the check results.

Algorithm 5.3: update_feature_attribute_metadata

```
Input: usr_schema, cdb_schema, cdb_bbox_type
Output: void: Populate the FEATURE_ATTRIBUTE_METADATA table from the result of
        querying existing feature attributes
1  Delete existing records of the specified cdb_schema;
2  Let $\mathbb{O} \leftarrow$ Array of existing objectclass_ids within the extent;
3  Let $\mathbb{N} \leftarrow$ datatype_ids of Nested attributes;
4  Let $\mathbb{A} \leftarrow$ datatype_id of the appearance property;
5  Let $\mathbb{D} \leftarrow$ datatype_id of the address property;
6  Let $\mathbb{F} \leftarrow$ datatype_id of the feature relation property;
7  Let $\mathbb{G} \leftarrow$ datatype_id of the geometry property;
8  Let $\mathbb{I} \leftarrow$ datatype_id of the implicit geometry property;
9  for each $o \in \mathbb{O}$ do
10     Let $o_d \leftarrow$ attribute datatype_id of $o$ ;
11     Query distinct Inline attributes for $o$ and exclude $o_d \in \{\mathbb{A}, \mathbb{D}, \mathbb{F}, \mathbb{G}, \mathbb{I}, \mathbb{N}\}$;
12     Query distinct pairs of Nested attributes for $o$ in the format of
         [parent_attri, child_attri] and exclude $o_d \in \{\mathbb{A}, \mathbb{D}, \mathbb{F}, \mathbb{G}, \mathbb{I}\}$;
13     Let $\alpha \leftarrow$ Array of Inline attributes;
14     Let $\beta \leftarrow$ Nested array of Nested attributes;
15     if $length(\alpha) \neq 0$ then
16         for each $\alpha_i \in \alpha$ do
17             Call check_feature_inline_attribute function with $o$ and $\alpha_i$;
18     if $length(\beta) \neq 0$ then
19         for each $\beta_i \in \beta$ do
20             Call check_feature_nested_attribute function with $o$ and $\beta_i$;
```

The check feature attribute metadata process starts with deleting the existing records in the FEATURE_ATTRIBUTE_METADATA table regarding the given cdb_schema, ensuring that the available feature attributes in the specified cdb_schema are up-to-date.

Since the cdb_schema PROPERTY table stores all feature properties, irrelevant attribute data types are required to be excluded from querying the distinct feature attributes regarding each available objectclass_id in the cdb_schema. These irrelevant attribute data types (datatype_id) include the address (8), appearance (9), feature relation (10), geometry (11), and implicit geometry properties (16) types. Note that the datatype_ids of the Nested attributes are checked by looking up the qgis_pkg ATTRIBUTE_DATATYPE_LOOKUP table, which is used to filter out Nested attributes when querying Inline attributes. The attribute datatype_ids from the "Dynamizer", "CityObjectGroup" and "Version" modules are excluded, and the query can be seen in Listing 5.21.

The update_feature_attribute_metadata first checks the irrelevant attribute datatype_ids and the available objectclass_ids within an extent in the cdb_schema. The query templates for `objectclass_id` availability check and extent selection (`sql_where`) are the same as Listing 5.6 and Listing 5.7. For every `objectclass_id`, the function then checks its available Inline and Nested attributes, and the query templates are shown in Listing 5.22 and Listing 5.23.

```sql
SELECT  STRING_AGG(adl.id::TEXT, ',') -- array as a string separated by comma
FROM qgis_pkg.attribute_datatype_lookup AS adl
WHERE adl.alias NOT IN ("dyn", "grp", "ver")
    AND adl.is_nested = 1; -- TRUE
```

Listing 5.21: Query for checking the `datatype_ids` of Nested attributes

```sql
SELECT
    ARRAY_AGG(DISTINCT p.name)
FROM ',qi_cdb_schema,'.feature AS f
    INNER JOIN ',qi_cdb_schema,'.property AS p ON (f.id = p.feature_id
        AND f.objectclass_id = ',oc_id,' ',sql_where,')
WHERE p.parent_id IS NULL
    -- Filter irrelevant attribute data types and Nested attribute
    AND p.datatype_id NOT IN (8, 9, 10, 11, 16, ',nested_attri_ids,');
```

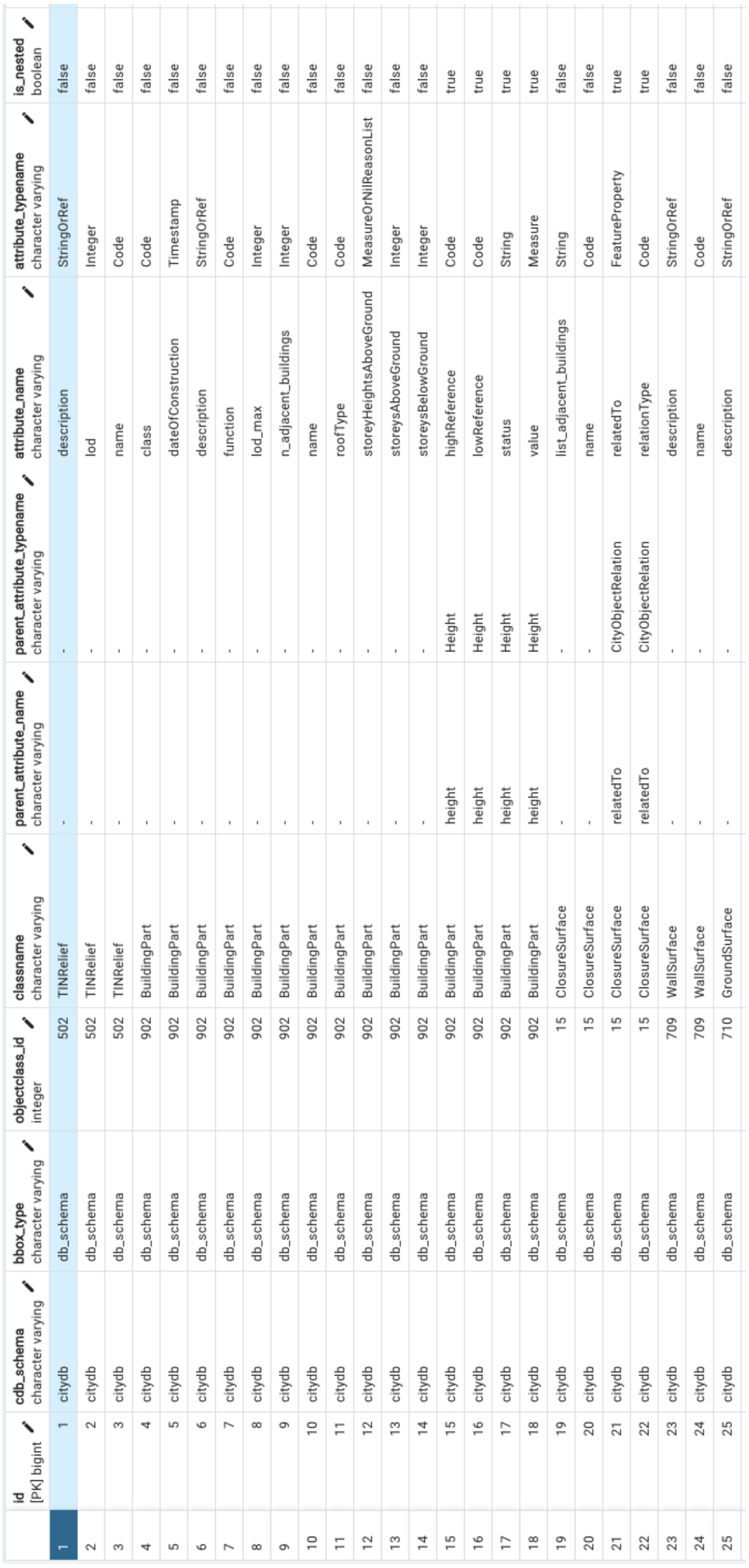
Listing 5.22: Query template for checking Inline attributes for each `objectclass_id`

```sql
SELECT
    ARRAY_AGG(ARRAY[p_attri, attri]) AS nested_attribute_set
FROM(
    SELECT DISTINCT p.name AS p_attri, p1.name AS attri
    FROM ',qi_cdb_schema,'.feature AS f
        INNER JOIN ',qi_cdb_schema,'.property AS p ON f.id = p.feature_id
            AND f.objectclass_id = ',oc_id,'
            -- Filter on parent attribute
            AND p.datatype_id NOT IN (8, 9, 10, 11, 16)',sql_where,'
        -- Join the child attributes and filter out the Inline attributes
        INNER JOIN ',qi_cdb_schema,'.property AS p1 ON p.id = p1.parent_id
    ORDER BY p.name, p1.name
) AS nested_attribute;
```

Listing 5.23: Query template for checking Nested attributes for each `objectclass_id`

The available Inline attributes `datatype_ids` are stored as an array. In contrast, the available Nested attributes `datatype_ids` are stored as a nested array, with each element in the format of parent and child attribute name tuples for referencing their hierarchical relations. Finally, the function traverses through these two attributes `datatype_ids` arrays and calls the `check_feature_inline_attribute` and `check_feature_nested_attribute` functions to populate the FEATURE_ATTRIBUTE_METADATA table.

The attribute meta table is populated by executing the insert values query text, generated dynamically based on each Inline attribute and Nested attribute pair. Several lookup functions such as `attribute_name_to_datatype_id` and `objectclass_id_to_classname` are called to check for the attribute `datatype_id` and class names, preparing the values for insertion. An example of the populated FEATURE_ATTRIBUTE_METADATA table is shown in Figure 5.8.



	id [PK] bigint	cdb_schema character varying	bbox_type character varying	objectclass_id integer	classname character varying	parent_attribute_name character varying	parent_attribute_typename character varying	attribute_name character varying	attribute_typename character varying	is_nested boolean
1	1	citydb	db_schema	502	TINRelief	-	-	description	StringOrRef	false
2	2	citydb	db_schema	502	TINRelief	-	-	lod	Integer	false
3	3	citydb	db_schema	502	TINRelief	-	-	name	Code	false
4	4	citydb	db_schema	902	BuildingPart	-	-	class	Code	false
5	5	citydb	db_schema	902	BuildingPart	-	-	dateOfConstruction	Timestamp	false
6	6	citydb	db_schema	902	BuildingPart	-	-	description	StringOrRef	false
7	7	citydb	db_schema	902	BuildingPart	-	-	function	Code	false
8	8	citydb	db_schema	902	BuildingPart	-	-	lod_max	Integer	false
9	9	citydb	db_schema	902	BuildingPart	-	-	n_adjacent_buildings	Integer	false
10	10	citydb	db_schema	902	BuildingPart	-	-	name	Code	false
11	11	citydb	db_schema	902	BuildingPart	-	-	roofType	Code	false
12	12	citydb	db_schema	902	BuildingPart	-	-	storeyHeightsAboveGround	MeasureOrNilReasonList	false
13	13	citydb	db_schema	902	BuildingPart	-	-	storeysAboveGround	Integer	false
14	14	citydb	db_schema	902	BuildingPart	-	-	storeysBelowGround	Integer	false
15	15	citydb	db_schema	902	BuildingPart	height	Height	highReference	Code	true
16	16	citydb	db_schema	902	BuildingPart	height	Height	lowReference	Code	true
17	17	citydb	db_schema	902	BuildingPart	height	Height	status	String	true
18	18	citydb	db_schema	902	BuildingPart	height	Height	value	Measure	true
19	19	citydb	db_schema	15	ClosureSurface	-	-	list_adjacent_buildings	String	false
20	20	citydb	db_schema	15	ClosureSurface	-	-	name	Code	false
21	21	citydb	db_schema	15	ClosureSurface	relatedTo	CityObjectRelation	relatedTo	FeatureProperty	true
22	22	citydb	db_schema	15	ClosureSurface	relatedTo	CityObjectRelation	relationType	Code	true
23	23	citydb	db_schema	709	WallSurface	-	-	description	StringOrRef	false
24	24	citydb	db_schema	709	WallSurface	-	-	name	Code	false
25	25	citydb	db_schema	710	GroundSurface	-	-	description	StringOrRef	false

Figure 5.8.: Populated FEATURE_ATTRIBUTE_METADATA table example

5.4.3. Attribute View Creation

After performing the attribute metadata scan, users can check the availability of feature attributes in the FEATURE_ATTRIBUTE_METADATA table. The main function used to create feature attribute views is named `create_attribute_view`, which is wrapped with other additional functions in the ”060_attribute” SQL file.

Algorithm 5.4 shows the pipeline of the `create_attribute_view` function. The function allows users to create views or materialized views of the selected feature attribute based on the existing feature classes to an extent. Users can specify the view types using the boolean parameter `is_matview`, and the corresponding query texts for view headers are generated by the same functions as mentioned in Section 5.3.3. The view headers are then added to the collect and flatten feature attribute queries for view creation. To enhance query time efficiency of the generated attribute materialized views, the `generate_sql_attribute_matview_footer` function stored in the ”060_attribute” SQL file is called to generate the footers to add indices dynamically based on the existing value columns.

The character varying parameter, `cdb_bbox_type`, can be specified for extent selection. The `is_matview` parameter is set to false by default for views while the `cdb_bbox_type` is set to ”db_schema” for full dataset extent by default.

Algorithm 5.4: `create_attribute_view`

```
Input: usr_schema, cdb_schema, objectclass_id ($o$), attribute_name ($a$),
       is_nested, cdb_bbox_type, is_matview
Output: void: Create feature attribute views or materialized views and store in
        usr_schema
1  Let $sql\_header \leftarrow$ generated based on the is_matview input;
2  Let $sql\_footer \leftarrow$ generated only when the is_matview is TRUE;
3  if not $is\_nested$ then
4      Call collect_inline_attribute function with $o$ and $a$;
5      Let $\mathbb{M} \leftarrow$ calculate the maximum attribute multiplicity;
6      if $\mathbb{M} > 1$ then
7          Let $\mathbb{V} \leftarrow$ check the number of value columns with;
8          $sql\_attri \leftarrow$ Call collect_inline_multiple_attribute function with $\mathbb{V}$;
9      else
10         Let $\mathbb{V} \leftarrow$ check the number of value columns with;
11         $sql\_attri \leftarrow$ Call collect_inline_single_attribute function with $\mathbb{V}$;
12 else
13     Call collect_nested_attribute function with $o$ and $a$;
14     Let $\mathbb{M} \leftarrow$ calculate the maximum attribute multiplicity;
15     if $\mathbb{M} > 1$ then
16         Let $\mathbb{V} \leftarrow$ check the number of value columns;
17         $sql\_attri \leftarrow$ Call collect_nested_multiple_attribute function with $\mathbb{V}$;
18     else
19         Let $\mathbb{V} \leftarrow$ check the number of value columns;
20         $sql\_attri \leftarrow$ Call collect_nested_single_attribute function with $\mathbb{V}$;
21 execute $(sql\_header \oplus sql\_attri \oplus sql\_footer)$;
```

The function first differentiates the Inline and Nested attribute classes based on the `is_nested` input. To flatten the selected attributes, the maximum attribute multiplicity check is performed in both the `collect_inline_attribute` and `collect_nested_attribute` functions. Listing 5.24 demonstrates the query template to check the maximum multiplicity of the selected attribute, where the number of the attribute entries regarding each `feature_id` is calculated by the `COUNT` and `GROUP BY` functions in the sub-query. The maximum counted number is obtained by performing the `MAX` function on the sub-query result. The attribute multiplicity check determines whether to use the crosstab function in the attribute flattening process (see Section 4.3.1.2).

```sql
SELECT
    MAX(count_p_name) AS max_count
FROM (
    SELECT f.id, f.objectclass_id, COUNT(p.name) AS count_p_name
    FROM ',qi_cdb_schema,'.property AS p
        INNER JOIN ',qi_cdb_schema,'.feature AS f ON (f.id = p.feature_id
            AND f.objectclass_id = ',oc_id,'
            AND p.parent_id IS NULL ',sql_where,')
    WHERE p.name = ',ql_attri_name,'
    GROUP BY f.id
) AS count_attri;
```

Listing 5.24: Query template for checking maximum attribute multiplicity

After the maximum attribute multiplicity check, the process is divided into four attribute classes: Inline-Single, Inline-Multiple, Nested-Single, and Nested-Multiple, which are introduced in Section 4.3.2. Four different functions are developed according to these attribute classes. Before flattening the feature attributes, the attribute value columns check is invoked to guarantee the existence of attribute values in the corresponding value columns. The attribute name is converted to the `datatype_id` by another function, and the target array of attribute value columns is retrieved by referencing the `qgis_pkg` ATTRIBUTE_DATATYPE_LOOKUP table with the attribute `datatype_id`. Listing 5.25 shows the query template for checking the attribute value existence of each value column from the column array. The attribute value column check gives the target columns for retrieving the attribute values. It determines the element columns in the composite types if the crosstab function is in use (see Section 4.3.1.3).

```sql
SELECT EXISTS(
    SELECT 1
    FROM ',qi_cdb_schema,'.feature AS f
        INNER JOIN ',qi_cdb_schema,'.property AS p ON f.id = p.feature_id
            AND f.objectclass_id = ',oc_id,'
    WHERE p.name = ',ql_attri_name,'
        AND p.',attri_val_col,' IS NOT NULL -- candidate value column name
    LIMIT 1);
```

Listing 5.25: Query template for checking attribute value columns

With the attribute multiplicity and value columns information checked, the function collects and flattens feature attributes to create attribute views or materialized views. The naming convention of the feature attribute view names is to concatenate the `cdb_schema` name, type labels, attribute class label, `objectclass_id`, and attribute name using underscores. The ”av” label is used for the attribute views, while the ”amv” label is for the attribute materialized views. The attribute class labels are "i" for the Inline attributes and "n" for the Nested attribute. If users choose to create the "height" attribute view of buildings from the "citydb" schema, the view name will be: `citydb_av_n_901_height`, as buildings' "height" is a Nested attribute. The attribute metadata, which are the input parameters of the `create_attribute_view` function, are used to create the view name dynamically within the function, and the view headers and footers are generated correspondingly.

Since the attribute views can have different numbers of columns, the attribute materialized view footer generation for adding indices consists of three steps: checking the candidate column names, checking the column types and creating the index. These three steps are achieved in three functions:

1. `get_view_column_name`: Given the attribute view name and `usr_schema`, returns an array of existing column names in an attribute view. The query template is shown in the Listing 5.26.

```sql
SELECT ARRAY (
    SELECT
        a.attname
    FROM pg_attribute AS a
        INNER JOIN pg_class AS t on a.attrelid = t.oid
        INNER JOIN pg_namespace s on t.relnamespace = s.oid
    WHERE a.attnum > 0
        AND NOT a.attisdropped
        AND t.relname = ',ql_view_name,'
        AND s.nspname = ',ql_usr_schema,'
ORDER BY a.attnum);
```

Listing 5.26: Query template for checking the column names in an attribute view

2. `get_view_column_type`: Given the attribute view name, `usr_schema` and attribute value column name, returns the attribute data type name of the input value column. This function avoids creating an index on the "json" type value column since it is not supported in PostgreSQL. The query template is shown in Listing 5.27.

```sql
SELECT FORMAT_TYPE(atttypid, atttypmod) AS data_type
FROM pg_attribute
WHERE attrelid = ',ql_view_name),' ::regclass
AND attname = ',ql_val_col_name,';
```

Listing 5.27: Query template for checking the data type of an attribute column

3. `generate_sql_attribute_matview_footer`: Get the existing attribute value columns with the above two functions and dynamically generate the attribute materialized view footer. An example query template is shown in Listing 5.28.

```sql
CREATE INDEX ',f_idx_name,' -- index name of feature ID
    ON ',qi_usr_schema,'.',qi_view_name,' (f_id);
------------------------------------------------------------------------------
-- Dynamically extended part
CREATE INDEX ',attri_idx_name,' -- index name of attribute column
    ON ',qi_usr_schema,'.',qi_view_name,' (',qi_view_col_name,');
------------------------------------------------------------------------------
ALTER TABLE ',qi_usr_schema,'.',qi_view_name,' OWNER TO ',qi_usr_name,';
```

Listing 5.28: Query template for attribute materialized view footer

The following subsections present the query templates for collecting and flattening Inline-Single, Inline-Multiple, Nested-Single, and Nested-Multiple attributes. The attribute metadata gathered from the FEATURE_ATTRIBUTE_METADATA table is passed to these templates by the `create_geometry_view` function for dynamically generating the SQL statements to collect and flatten feature attributes. The attribute value column check function indicated in Listing 5.25 retrieves the target attribute source value column. The `sql_where` query can also be added in the query to perform extent selection.

5.4.3.1. Collect and Flatten Inline-Single Attributes

The `collect_inline_single_attribute` function is invoked in this case. Listing 5.29 shows the query template to collect and flatten the Inline-Single attributes. The function checks the number of target attribute value columns and dynamically renames the resulting attribute value columns to the attribute name.

For Inline-Single attributes with one value column, the value column name is placed in the `SELECT` clause and renamed to the target attribute name. In the case of multiple value columns, the first value column is renamed to the target attribute name while the prefix `val_` in the remaining value columns is replaced by the target attribute name. An example of this attribute class is the "class" attribute of buildings, and the generated query and result are shown in Listing 4.6 and Figure 4.15.

```sql
SELECT
    f.id AS f_id,
    p.',qi_attri_val_col_1,' AS ',attri_name,',
    --------------------------------------------------------------------------
    -- Dynamically extended part
    p.',qi_attri_val_col_2,'
        AS ',qi_attri_name, '_','SUBSTRING(',qi_attri_val_col,' FROM val_(.*))
    ...
    --------------------------------------------------------------------------
FROM citydb.feature AS f
    INNER JOIN citydb.property AS p ON f.id = p.feature_id
        AND f.objectclass_id = 'oc_id'  -- Target feature class
        AND p.name = 'qi_attri_name'; -- Target attribute
        ',sql_where,'
```

Listing 5.29: Query template for collecting and flattening Inline-Single attributes

5.4.3.2. Collect and Flatten Inline-Multiple Attributes

The `collect_inline_multiple_attribute` function generates the query of the Inline-Multiple attributes differently based on the number of value columns. The differences in the query are explained structurally to satisfy the use of the crosstab function.

- **Composite type header**
  The composite type is created in the case of multiple value columns, combining all value columns into a single composed value column for the crosstab function. The composite type header creation template is shown in Listing 5.30. The definition wraps each attribute value column with its data type separated by a space. The definition of creation type is generated dynamically based on the input attribute value columns array.

```sql
DROP TYPE IF EXISTS ',qi_ct_type_name,';
---------------------------------------------------------------------------
-- Dynamically extended part
CREATE TYPE ',qi_ct_type_name,'
    AS (',qi_attri_val_col_1,' ',qi_attri_val_col_1_type,',
        ',qi_attri_val_col_2,' ',qi_attri_val_col_2_type,',
        ...);
---------------------------------------------------------------------------
```

Listing 5.30: Query template of the composite type header for Inline-Multiple attributes

- **Crosstab SELECT clause**
  The crosstab `SELECT` clause is determined by the number of value columns. Listing 5.31 provides the template for one value column, where the value columns are renamed to the attribute name followed by the multiplicity count. This process repeats to extend the query until the value column with the maximum multiplicity number is renamed.

  Listing 5.32 shows the template for multiple value columns, where the actual values of each attribute entry are extracted from the renamed composite-type value tuples using dot notation according to its source value column name. The first value column of each attribute entry is renamed to the attribute name with the multiplicity count. In contrast, the remaining value columns are renamed to the attribute name followed by the source value column suffix and the multiplicity count. The source value column suffixes are retrieved using the `SUBSTRING` function to slice the source value column name. This process repeats to extend the query until the actual attribute values and the value columns of the attribute entry with the maximum multiplicity number are extracted and renamed.

```sql
SELECT
    f_id AS f_id,
    ----------------------------------------------------------------------
    -- Dynamically extended part
    ct.',qi_attri_name,'_',multi_count,' AS ',qi_attri_name,'_',multi_count,',
    ct.',qi_attri_name,'_',multi_count,' AS ',qi_attri_name,'_',multi_count,'
    ...
    ----------------------------------------------------------------------
```

Listing 5.31: Query template of the crosstab `SELECT` clause for Inline-Multiple attributes (One attribute value column)

```sql
SELECT
    f_id AS f_id,
    ----------------------------------------------------------------------
    -- Dynamically extended part
    (',qi_attri_name,'_',multi_count,').',qi_attri_val_col_1,'
        AS ',qi_attri_name,'_',multi_count,',

    (',qi_attri_name,'_',multi_count,').',qi_attri_val_col_2,'
        AS ',qi_attri_name,'_',
           'SUBSTRING(',qi_attri_val_col,' FROM val_(.*))', '_', multi_count,'
    ...
    ----------------------------------------------------------------------
```

Listing 5.32: Query template of the crosstab `SELECT` clause for Inline-Multiple attributes (Multiple attribute value columns)

- **Crosstab FROM clause**
  The crosstab `FROM` clause is determined by the number of value columns. Listing 5.33 provides the template for one value column, where the crosstab resulting value columns are renamed to the attribute name followed by the multiplicity count, and they are cast to the source value column data type. This process repeats to extend the query until all value columns are renamed and typecast up to the maximum multiplicity number.

  Listing 5.34 shows the template for multiple value columns, where the source value columns are composed dynamically and cast to the composite type in the crosstab input SQL statement. In the crosstab resulting value columns definition, the value column of each attribute entry is renamed to the attribute name with the multiplicity count and cast to the composite type. This process repeats to extend the query until the value column of the attribute entry with the maximum multiplicity number is renamed and typecast.

```sql
FROM CROSSTAB($BODY$
    SELECT
        f.id AS f_id,
        p.name,
        p.',qi_attri_val_col,'
    FROM citydb.feature AS f
        INNER JOIN citydb.property AS p ON (f.id = p.feature_id
            AND f.objectclass_id = ',oc_id,' -- Target class
            ',sql_where,')
    WHERE p.name = ',qi_attri_name,' -- Target attribute
    ORDER BY f_id, p.id ASC $BODY$)
    AS ct(f_id bigint,
        ------------------------------------------------------------------
        -- Dynamically extended part
         ',qi_attri_name,'_',multi_count,' ',qi_attri_val_col_type,',
         ',qi_attri_name,'_',multi_count,' ',qi_attri_val_col_type,'
         ...);
        ------------------------------------------------------------------
```

Listing 5.33: Query template of the crosstab `FROM` clause for Inline-Multiple attributes (One attribute value columns)

```
FROM CROSSTAB($BODY$
    SELECT
        f.id AS f_id,
        p.name,
        ------------------------------------------------------------------
        -- Dynamically extended part
        (p.',qi_attri_val_col_1,',
         p.',qi_attri_val_col_2,' ... )::',qi_ct_type_name,'
        ------------------------------------------------------------------
    FROM citydb.feature AS f
        INNER JOIN citydb.property AS p ON (f.id = p.feature_id
            AND f.objectclass_id = ',oc_id,' -- Target class
            ',sql_where,')
    WHERE p.name = ',attri_name,' -- Target attribute
    ORDER BY f_id, p.id ASC $BODY$)
    AS ct(f_id bigint,
        ------------------------------------------------------------------
        -- Dynamically extended part
        ',qi_attri_name,'_',multi_count,' ',qi_ct_type_name,',
        ',qi_attri_name,'_',multi_count,' ',qi_ct_type_name,'
        ...);
        ------------------------------------------------------------------
```

Listing 5.34: Query template of the crosstab `FROM` clause for Inline-Multiple attributes (Multiple attribute value columns)

The output query of `collect_inline_multiple_attribute` function is generated dynamically by consisting of the above three main parts based on the number of source attribute value columns. Examples of this attribute class are the ”name” (one value column) and ”function” (multiple value columns) attributes of buildings, and the generated queries are shown in Listing 4.7 and Listing 4.8. At the same time, the results can be seen in Figure 4.16 and Figure 4.17.

5.4.3.3. Collect and Flatten Nested-Single Attributes

The `collect_nested_single_attribute` function is called to generate the query dynamically for collecting and flattening Nested-Single attributes. The child attributes of the target parent attribute are gathered in an array by looking up the FEATURE_ATTRIBUTE_METADATA table, as shown in Listing 5.35. The function traverses through the child attributes array, checks the value columns for each child attribute, and generates the `SELECT` clause dynamically.

Listing 5.36 shows the query template to collect and flatten the Nested-Single attributes. In the `FROM` clause, the `cdb_schema` FEATURE table is joined with the PROPERTY table twice, where the first join queries for the parent attribute. In the `SELECT` clause, the `CASE` and `MAX` functions are applied to handle each child attribute with its attribute value column name specified after the alias of the second joined `cdb_schema` PROPERTY table (p1). Every resulting value column is renamed to the combination of parent-child attribute names. This process repeats to extend the query until all child attributes are queried and renamed. An example of this attribute class is the ”height” attribute of buildings. The generated query and result are shown in Listing 4.9 and Figure 4.18.

```sql
SELECT ARRAY_AGG(attribute_name)
FROM qgis_pkg.feature_attribute_metadata
WHERE cdb_schema = ',qi_cdb_schema,'
    AND objectclass_id = ',oc_id,'
    AND parent_attribute_name = ',qi_parent_attri_name,';
```

Listing 5.35: Query template for gathering child attributes of the given parent attribute

```sql
SELECT
    f.id AS f_id,
    ----------------------------------------------------------------------
    -- Dynamically extended part
    MAX(CASE WHEN p1.name = ',qi_child_attri_name_1,'
        THEN p1.',qi_child_attri_val_col_1,' END)
            AS ',qi_parent_attri_name,'_',qi_child_attri_name_1,',
    MAX(CASE WHEN p1.name = ',qi_child_attri_name_2,'
        THEN p1.',qi_child_attri_val_col_2,' END)
            AS ',qi_parent_attri_name,'_',qi_child_attri_name_2,',
    ...
    ----------------------------------------------------------------------
FROM citydb.feature AS f
    INNER JOIN citydb.property AS p ON (f.id = p.feature_id
        AND f.objectclass_id = ',oc_id,' -- Target class
        AND p.name = ',qi_parent_attri_name,') -- Target attribute
    INNER JOIN citydb.property AS p1 ON p.id = p1.parent_id
GROUP BY f.id;
```

Listing 5.36: Query template for collecting and flattening Nested-Single attributes

5.4.3.4. Collect and Flatten Nested-Multiple Attributes

The `collect_nested_multiple_attribute` function is used to generate the query dynamically for collecting and flattening Nest-Multiple attributes. In this attribute class, the implementation of the Inline-Multiple attributes for using crosstab and composite type (Section 5.4.3.2) and Nested-Single attribute for renaming the resulting value columns (Section 5.4.3.3) are combined. The child attributes of the target parent attribute are gathered in an array using the same query shown in Listing 5.35. The function traverses through the child attributes array, checks the value columns for each child attribute, and generates the crosstab `SELECT` and `FROM` clauses dynamically. The query is explained structurally to satisfy the use of the crosstab function.

- **Composite type header**
  The child attributes of this attribute class usually have their values stored across multiple value columns. Thus, the composite type must combine all value columns into a single composed value column for the crosstab function. Listing 5.37 shows the composite type header template. The child attribute value columns are wrapped with their data types separated by spaces, and the composite type definition is extended dynamically based on the existing child attribute value columns.

```
DROP TYPE IF EXISTS ',qi_ct_type_name,';
---------------------------------------------------------------------------
-- Dynamically extended part
CREATE TYPE ',qi_ct_type_name,'
    AS (',qi_child_attri_val_col_1,' ',qi_child_attri_val_col_1_type,',
        ',qi_child_attri_val_col_2,' ',qi_child_attri_val_col_2_type,',
        ...);
---------------------------------------------------------------------------
```

Listing 5.37: Query template of the composite type header for Nested-Multiple attributes

- **Crosstab SELECT clause**
  Listing 5.38 shows the crosstab `SELECT` clause template for the Nested attributes, where the actual values of each child attribute entry are extracted from the renamed composite-type value tuples using dot notation according to its source value column name. The resulting value columns of each Nested parent attribute entry are renamed to the parent-child attribute name combinations followed by the multiplicity count. If a child attribute has multiple source value columns, its resulting value column name is renamed to the parent-child attribute name combinations followed by the source value column suffix and the multiplicity count. This process repeats to extend the query until the actual child attribute values and the resulting value columns of the Nested attribute entry with the maximum multiplicity number are extracted and renamed.

```
SELECT
    f_id AS f_id,
    -----------------------------------------------------------------------
    -- Dynamically extended part
    (',qi_child_attri_name_1,'_',multi_count,').',qi_attri_val_col_1,'
        AS ',qi_child_attri_name_1,'_',multi_count,',

    (',qi_child_attri_name_2,'_',multi_count,').',qi_attri_val_col_2,'
        AS ',qi_child_attri_name_2,'_',
            'SUBSTRING(',qi_child_attri_val_col_2,' FROM val_(.*))', '_',
            multi_count,'
    ...
    -----------------------------------------------------------------------
```

Listing 5.38: Query template of the crosstab `SELECT` clause for Nested-Multiple attributes

- **Crosstab FROM clause**
  Listing 5.38 shows the crosstab `FROM` clause template for the Nested attributes, where the child attribute source value columns are composed dynamically and cast to the composite type in the crosstab input SQL statement. In the crosstab resulting value columns definition, the value columns for each Nested parent attribute entry are renamed to the child attribute names followed by the multiplicity count. These renamed columns are then cast to the composite type. This process repeats to extend the query until the child value columns of the Nested parent entry with the maximum multiplicity number are renamed and typecast.

```
FROM CROSSTAB($BODY$
    SELECT
        f.id AS f_id,
        p.name,
        ------------------------------------------------------------------
        -- Dynamically extended part
        (p.',qi_child_attri_val_col_1,',
         p.',qi_child_attri_val_col_2,' ... )::',qi_ct_type_name,'
        ------------------------------------------------------------------
    FROM citydb.feature AS f
        INNER JOIN citydb.property AS p ON (f.id = p.feature_id
            AND f.objectclass_id = ',oc_id,' -- Target class
            AND p.name = ',qi_parent_attri_name,' -- Target attribute
            ',sql_where,')
        INNER JOIN citydb.property AS p1 ON p.id = p1.parent_id
    ORDER BY f.id, p.id, p1.name ASC $BODY$)
    AS ct(f_id bigint,
        ------------------------------------------------------------------
        -- Dynamically extended part
        ',qi_child_attri_name_1,'_',multi_count,' ',qi_ct_type_name,',
        ',qi_child_attri_name_2,'_',multi_count,' ',qi_ct_type_name,'
        ...);
        ------------------------------------------------------------------
```

Listing 5.39: Query template of the crosstab `FROM` clause for Nested-Multiple attributes

The `collect_Nested_multiple_attribute` function dynamically generates the query using the above three parts. Examples of this attribute class can be the ”height” attributes of buildings. The generated query and result are shown in Listing 4.10 and Figure 4.19.

5.4.3.5. Attribute View Results

The `create_attribute_view` function uses the attribute metadata retrieved from the FEATURE_ATTRIBUTE_METADATA table as inputs (see Algorithm 5.4) to dynamically generate SQL statements based on the templates introduced above, which are executed to create feature (flattened) attribute views or materialized views. Listing 5.40 demonstrates examples of creating a view of buildings' ”description” (Inline) attribute and a materialized view of buildings' ”height” (Nested) attribute. The attribute materialized view is refreshed when created. The created results are stored in the `usr_schema` named ”qgis_bstsai”, and the view names and other metadata regarding the views are updated in the attribute metadata table.

The created attribute views consist of `f_id` and all existing attribute value columns based on the selected attribute, where the first column is stored to be associated with the feature geometries. Users can check the generated attribute views and materialized views either in pgAdmin4 or in QGIS. The generated results of the four attribute classes can be seen in Figure 5.9 (Inline-Single attribute), Figure 5.10 (Inline-Multiple attribute), Figure 5.11 (Nested-Single attribute) and Figure 5.12 (Nested-Multiple attribute).

```sql
-- View: buildings' description
SELECT qgis_pkg.create_attribute_view('qgis_bstsai', 'citydb',
                                       901, 'description');
-- Materialized View: buildings' height
SELECT qgis_pkg.create_attribute_view('qgis_bstsai', 'citydb',
                     901, 'height', 'TRUE', 'db_schema', TRUE);
```

Listing 5.40: Calling the `create_attribute_view` function

Additional functions for users to manage the feature attribute views are included in the "060_attribute" SQL file. For instance, the `create_all_attribute_view_in_schema` function allows users to bulk-create views of available feature attributes within an extent from the selected `cdb_schema`. Listing 5.41 shows two modes for bulk creation of views. Users can specify the target class to create attribute views of that class or use the default inputs to create all available feature attribute views.

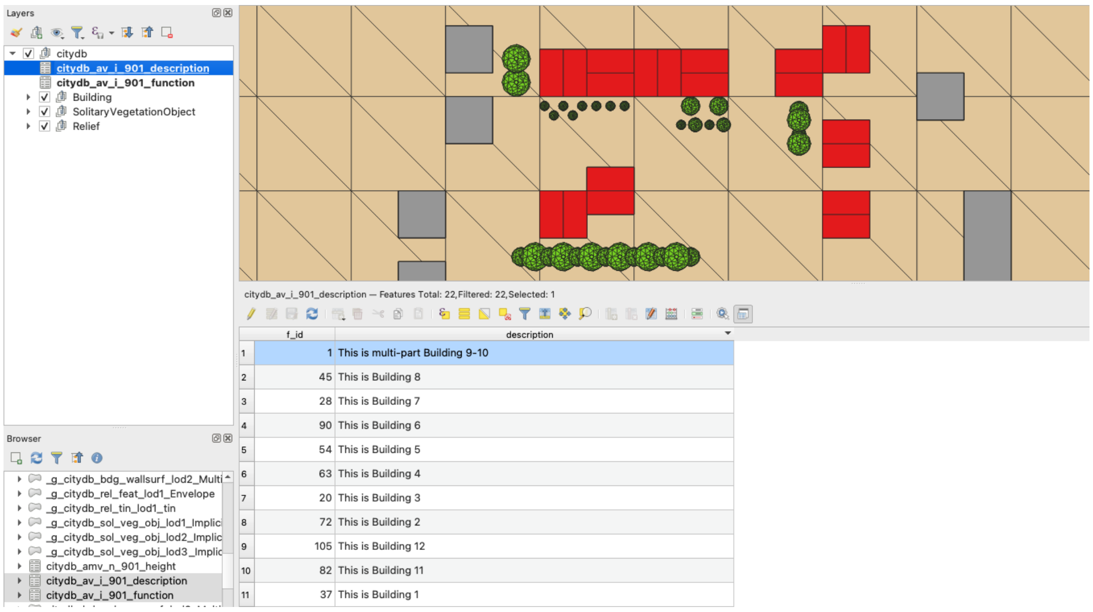
```sql
-- Inputs (default value): usr_schema, cdb_schema, oc_id (NULL), is_matview (
    FALSE), cdb_bbox_type ('db_schema')
-- All classes
SELECT qgis_pkg.create_all_attribute_view_in_schema
    ('qgis_bstsai', 'citydb'); -- view
SELECT qgis_pkg.create_all_attribute_view_in_schema
    ('qgis_bstsai', 'citydb', NULL, TRUE); -- materialized view
-- Building class only
SELECT qgis_pkg.create_all_attribute_view_in_schema
    ('qgis_bstsai', 'citydb', 901); -- view
SELECT qgis_pkg.create_all_attribute_view_in_schema
    ('qgis_bstsai', 'citydb', 901, TRUE); -- materialized view
```

Listing 5.41: Calling the `create_all_attribute_view_in_schema` function

Finally, the generated attribute views can be dropped individually or jointly by calling the `drop_all_attribute_views` function. Listing 5.42 shows two modes to drop the generated geometry views. The metadata of the view names and the view creation date are removed from the FEATURE_ATTRIBUTE_METADATA table.

```sql
-- Inputs (default value): usr_schema, cdb_schema, oc_id (NULL)
-- All classes
SELECT qgis_pkg.drop_all_attribute_views('qgis_bstsai', 'citydb');
-- Building class only
SELECT qgis_pkg.drop_all_attribute_views('qgis_bstsai', 'citydb', 901);
```

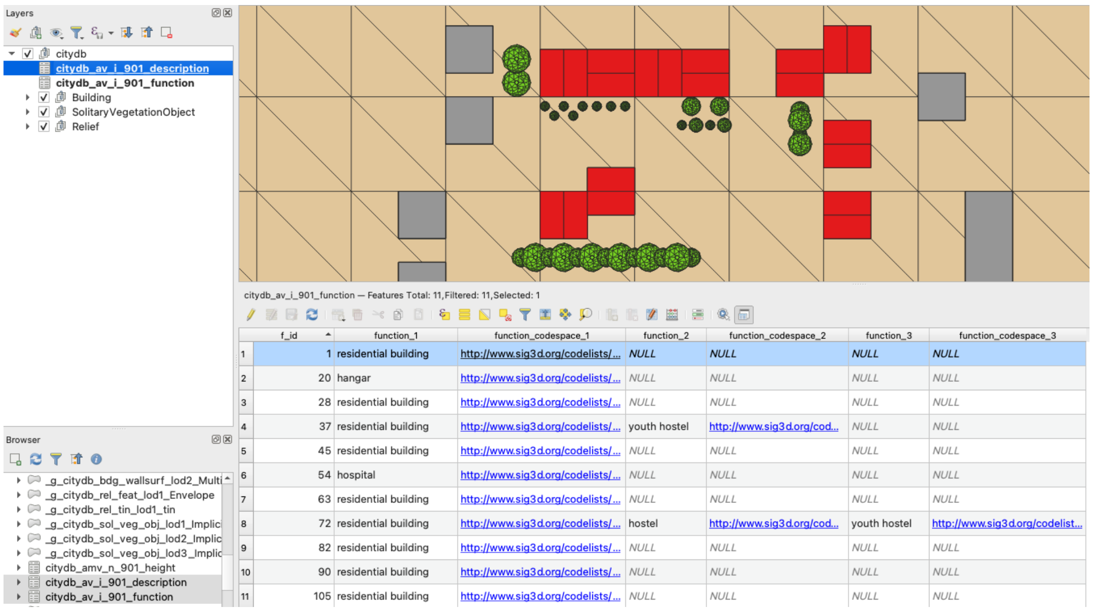
Listing 5.42: Calling the `drop_all_attribute_view` function


Layers
citydb
citydb_av_i_901_description
citydb_av_i_901_function
Building
SolitaryVegetationObject
Relief
Browser
_g_citydb_bdg_wallsurf_lod2_Multi
_g_citydb_rel_feat_lod1_Envelope
_g_citydb_rel_tin_lod1_tin
_g_citydb_sol_veg_obj_lod1_Implic
_g_citydb_sol_veg_obj_lod2_Implic
_g_citydb_sol_veg_obj_lod3_Implic
citydb_amv_n_901_height
citydb_av_i_901_description
citydb_av_i_901_function

citydb_av_i_901_description — Features Total: 22,Filtered: 22,Selected: 1

	f_id	description
1	1	This is multi-part Building 9-10
2	45	This is Building 8
3	28	This is Building 7
4	90	This is Building 6
5	54	This is Building 5
6	63	This is Building 4
7	20	This is Building 3
8	72	This is Building 2
9	105	This is Building 12
10	82	This is Building 11
11	37	This is Building 1

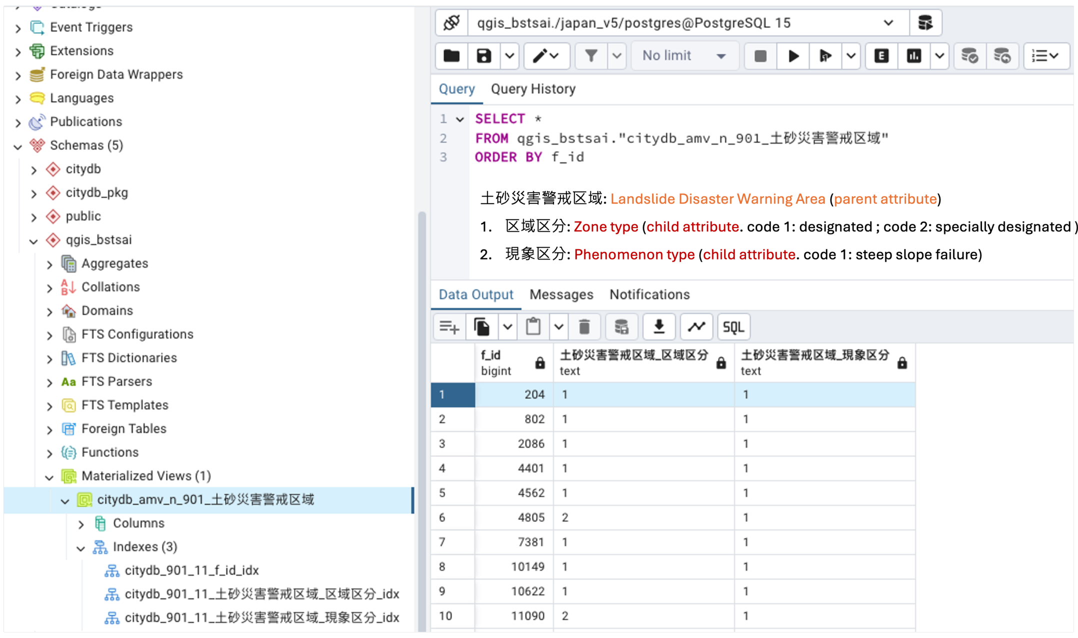
Figure 5.9.: Inline-Single attribute view checked in QGIS - "description" of buildings from the Alderaan dataset


Layers
citydb
citydb_av_i_901_description
citydb_av_i_901_function
Building
SolitaryVegetationObject
Relief
Browser
_g_citydb_bdg_wallsurf_lod2_Multi
_g_citydb_rel_feat_lod1_Envelope
_g_citydb_rel_tin_lod1_tin
_g_citydb_sol_veg_obj_lod1_Implic
_g_citydb_sol_veg_obj_lod2_Implic
_g_citydb_sol_veg_obj_lod3_Implic
citydb_amv_n_901_height
citydb_av_i_901_description
citydb_av_i_901_function

citydb_av_i_901_function — Features Total: 11,Filtered: 11,Selected: 1

	f_id	function_1	function_codespace_1	function_2	function_codespace_2	function_3	function_codespace_3
1	1	residential building	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
2	20	hangar	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
3	28	residential building	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
4	37	residential building	http://www.sig3d.org/codelists/...	youth hostel	http://www.sig3d.org/cod...	NULL	NULL
5	45	residential building	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
6	54	hospital	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
7	63	residential building	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
8	72	residential building	http://www.sig3d.org/codelists/...	hostel	http://www.sig3d.org/cod...	youth hostel	http://www.sig3d.org/codelist...
9	82	residential building	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
10	90	residential building	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL
11	105	residential building	http://www.sig3d.org/codelists/...	NULL	NULL	NULL	NULL

Figure 5.10.: Inline-Multiple attribute view checked in QGIS - "function" of buildings from the Alderaan dataset


qgis_bstsai./japan_v5/postgres@PostgreSQL 15
Materialized Views (1)
citydb_amv_n_901_土砂災害警戒区域
Indexes (3)
citydb_901_11_f_id_idx
citydb_901_11_土砂災害警戒区域_区域区分_idx
citydb_901_11_土砂災害警戒区域_現象区分_idx

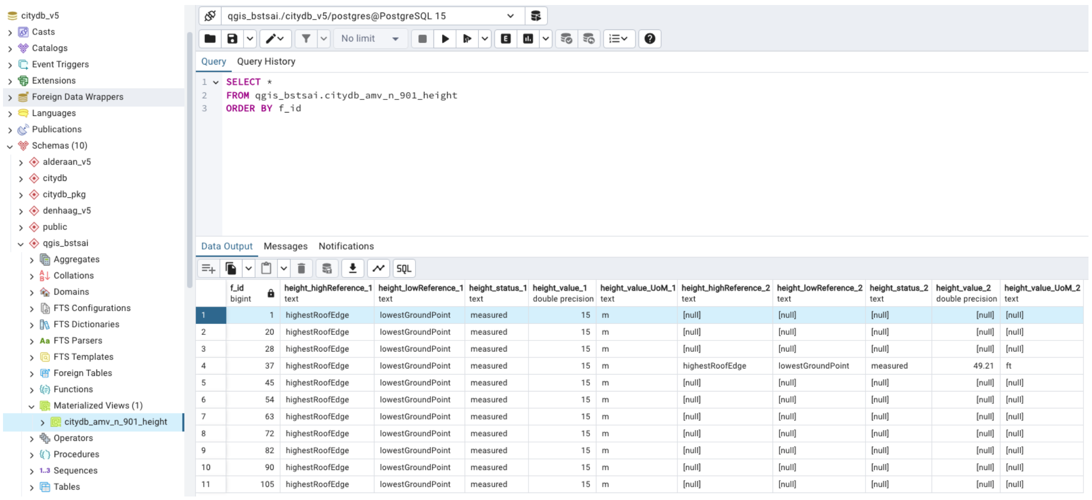
```
SELECT *
FROM qgis_bstsai."citydb_amv_n_901_土砂災害警戒区域"
ORDER BY f_id
```

土砂災害警戒区域: Landslide Disaster Warning Area (parent attribute)

1. 区域区分: Zone type (child attribute. code 1: designated ; code 2: specially designated)
2. 現象区分: Phenomenon type (child attribute. code 1: steep slope failure)

	f_id bigint	土砂災害警戒区域_区域区分 text	土砂災害警戒区域_現象区分 text
1	204	1	1
2	802	1	1
3	2086	1	1
4	4401	1	1
5	4562	1	1
6	4805	2	1
7	7381	1	1
8	10149	1	1
9	10622	1	1
10	11090	2	1

Figure 5.11.: Nested-Single attribute materialized view checked in pgAdmin4 - "Landslide Disaster Warning Area" of buildings from the Tokyo dataset (Code values translated from [46])



```
SELECT *
FROM qgis_bstsai.citydb_amv_n_901_height
ORDER BY f_id
```

	f_id bigint	height_highReference_1 text	height_lowReference_1 text	height_status_1 text	height_value_1 double precision	height_value_UoM_1 text	height_highReference_2 text	height_lowReference_2 text	height_status_2 text	height_value_2 double precision	height_value_UoM_2 text
1	1	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
2	20	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
3	28	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
4	37	highestRoofEdge	lowestGroundPoint	measured	15	m	highestRoofEdge	lowestGroundPoint	measured	49.21	ft
5	45	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
6	54	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
7	63	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
8	72	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
9	82	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
10	90	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]
11	105	highestRoofEdge	lowestGroundPoint	measured	15	m	[null]	[null]	[null]	[null]	[null]

Figure 5.12.: Nested-Multiple attribute materialized view checked in pgAdmin4 - "height" of buildings from the Alderaan dataset

5.4.4. Lessons Learned from the Feature Attributes

The integrated characteristic and the Nested attributes of 3DCityDB v.5.0 encoding present a more challenging process for flattening (linearising) feature attributes, unlike the direct retrieval of querying the flattened attributes encoded with 3DCityDB v.4.x. Compared to 3DCityDB v.4.x, all types of feature attributes, including general, generic, and specific attributes, require flattening to be joined with feature geometries. The Inline-Nested attribute classes, attribute multiplicity, and the number of attribute value columns are the three crucial factors that lead to the four attribute flattening approaches: Inline-Single, Inline-Multiple, Nested-Single, and Nested-Multiple. Attribute multiplicity determines the use of the crosstab function, while the number of attribute value columns necessitates using composite types when the crosstab function is in use.

The server-side `qgis_pkg` developed by this research allows users to select their desired feature attribute for each class in a CityGML module, offering more flexibility than the current plug-in. The main differences include:

- **Current plug-in (v.0.8.9)**
  - General and specific attributes are not selectable by users. These attributes are linked with feature geometries when creating GIS layers.
  - Generic attributes are processed as sub-tables and associated with GIS layers. Users can only check the generic attributes individually.
- **Server-side `qgis_pkg` of this research**
  - All attributes are flattened (linearised), allowing users to customise which attributes are joined with feature geometries for creating GIS layers.
  - Generic attributes are flattened as ”normal” attributes, enabling users to check them with multiple features.
  - Attribute names in other alphabets (e.g. Kanji in Japanese) can be used as column names of GIS layers.

For example, users can choose to create attribute views of ”description”, ”function” and ”height” of buildings and join them with feature geometries, whereas the current plug-in links all available general and specific attributes of buildings with the selected buildings' geometries. The selectable attribute linking enhances user interaction with the abundant CityGML data stored in 3DCityDB.

6. Implementation Part 2: GIS Layer Creation

This chapter introduces approaches to join feature geometries and attributes for creating GIS layers. According to Section 4.4, feature geometry materialized views are joined with user-selected attribute views to create GIS layers. Since feature attributes are selectable based on users' decisions, three approaches for joining these attributes with geometry materialized views are proposed to assess the query performance of the generated GIS layers.

Section 6.1 elaborates on the pipeline of GIS layer creation, which includes sub-sections referring to three different approaches. Section 6.1.1 introduces the first approach, where user-selected feature attributes are stored in individual views. These attribute views are then associated with the specified feature geometry materialized view using the `LEFT JOIN` function based on the `f_id`. Section 6.1.2 provides a similar joining process, with the difference that user-selected attributes are stored in individual materialized views. Section 6.1.3 presents an integrated version of attribute materialized views, where the feature attributes chosen by users are first joined together using the `FULL JOIN` function and stored in a materialized view. This integrated attribute view table is joined with the specified feature geometry materialized view. Section 6.2 shows the query performance test to check the query time from the generated GIS layers under specified conditions. Finally, the GIS layer creation results are presented in Section 6.4.

6.1. Layer Creation Approaches

The PL/pgSQL function used to create GIS layers is the `create_layer` function. This function calls one of three sub-functions based on the provided inputs: `create_layer_multiple_join`, `create_layer_multiple_join_all_attri`, and `create_layer_attri_table`. The first and second sub-functions correspond to approaches 1 and 2, while the third corresponds to approaches 3. For the target feature geometry representation and LoD, the `id` of the target geometry within the FEATURE_GEOMETRY_METADATA table is derived from the user's specification using the `get_geometry_key_id` function. In contrast, the `ids` of target attributes within the FEATURE_ATTRIBUTE_METADATA table are obtained via the `get_attribute_key_id` function, which takes the desired attribute names in an array as its input. All functions involved in the GIS layer creation are included in the ”070_layers” SQL file.

Algorithm 6.1 illustrates the pipeline of how these three functions are invoked within the `create_layer` function, based on three selectable parameters. The parameter `is_matview` determines whether views or materialized views are used to store the selected attributes and specifies the output view types of layers. The parameter `is_all_attri` indicates whether all available attributes are chosen based on the provided `objectclass_id`, and triggers an attributes scan to generate an array of all existing attribute `ids` if `is_all_attri` is set to `TRUE`. Finally, `is_joins` decides which approach to create the layers. The function starts with converting geometry and attributes `ids`. The `is_joins` parameter first determines which approach is applied for layer creation, and`is_all_attri` dictates whether all attributes are included in the layers generated by multiple joins between individual attribute views or materialized views with the specified geometry.

Algorithm 6.1: `create_layer`

```
Input: usr_schema, cdb_schema, parent_objectclass_id ($po$), objectclass_id ($o$),
       geometry_name ($g$), attributes ($attris$), is_matview, is_all_attri, is_joins
Output: void: Joining the specified feature LoD, geometry representation, and
        attributes to create GIS layers
1  Let $\mathbb{G} \leftarrow$ id of feature geometry representation and LoD retrieved by the
   get_geometry_key_id function with $po, o$, and $g$;
2  if $attris \neq$ NULL and not $is\_all\_attri$ then
3      for each $attri \in attris$ do
4          Let $\mathbb{A} \leftarrow$ Array of feature attributes ids retrieved by the
           get_attribute_key_id function with $o$ and $attri$;
5  if not $is\_joins$ then
6      Call create_layer_attri_table function with $\mathbb{G}$ and $\mathbb{A}$;
7  else
8      if $is\_all\_attri$ then
9          Call create_layer_multiple_joins_all_attri function with $\mathbb{G}$ and $o$;
10     else
11         Call create_layer_multiple_joins function with $\mathbb{G}$ and $\mathbb{A}$;
```

The naming convention of the GIS layers is to concatenate indicator labels, such as ”=lv” for ”layer view” and ”=lmv” for ”layer materialized view,” with the `cdb_schema` name, class alias, geometry, and attribute labels using underscores. The attribute labels differ based on different modes in different approaches:

- **Approach 1 & 2** (`is_joins` set to `TRUE`)
    - **Select mode**: This mode creates labels if users choose only to join certain attributes. The ”ia” label represents the Inline attribute, while the ”na” label represents the Nested attribute. The attribute `id` numbers, enclosed in curly brackets, are then attached after the labels using underscores. For example, if the `ids` of the building ”function” and ”height” are 42 and 57, the attribute label will be ”_ia_{42}_na_{57}”.
    - **All attributes mode**: The label will be ”_all_attri_joins”.
    - **No attributes mode**: The label will be ”no_attris_joins”.
- **Approach 3** (`is_joins` set to `FALSE`): The attribute label generated in this approach is consistently set to ”_attri_table” regardless of the attribute selection modes, to avoid auto-truncation of the attribute table name in PostgreSQL. If no attributes are selected using approach 3, the attribute label is set to ”no_attris_table”.

If users choose to create a layer using approach 2 with the buildings in LoD1 Solid and only want to inspect the ”name,” ”function,” and ”height” attributes of buildings from the ”citydb” schema, the view name would be ”`=lmv_citydb_bdg_lod1_Solid_ia_{42,46}_na_{57}`”, as approach two stores the result in a materialized view, and the "name" and "function" of buildings are Inline attributes while "height" is a Nested attribute. The geometry and attribute metadata tables serve as menus for users to customise the GIS layer creation based on their needs. Similar to the feature geometry and attribute views creation, the user specifications are the input parameters of the `create_layer` function, where the view headers and footers are generated correspondingly to create layers dynamically.

The following three sub-sections propose three different approaches considering view types for storing the selected attributes and the implementation to associate them with target feature geometry materialized view to create GIS layers.

6.1.1. Approach 1- Joining Feature Geometry Materialized Views and Attribute Views

In approach 1, the target feature geometry materialized view is associated with the selected attribute views for layer creation. If the view of a selected attribute has not been created, the `create_layer` function calls the `create_attribute_view` function to create it first. Listing 6.1 shows the query template of approach 1, where every value column specified after the attribute number alias, except the `f_id` in each selected attribute view, is dynamically extracted and inserted into the `SELECT` clause. In the `FROM` clause, the `LEFT JOIN` function is performed repetitively to join geometries with selected attributes on `f_id` and extended dynamically based on the number of selected attributes.

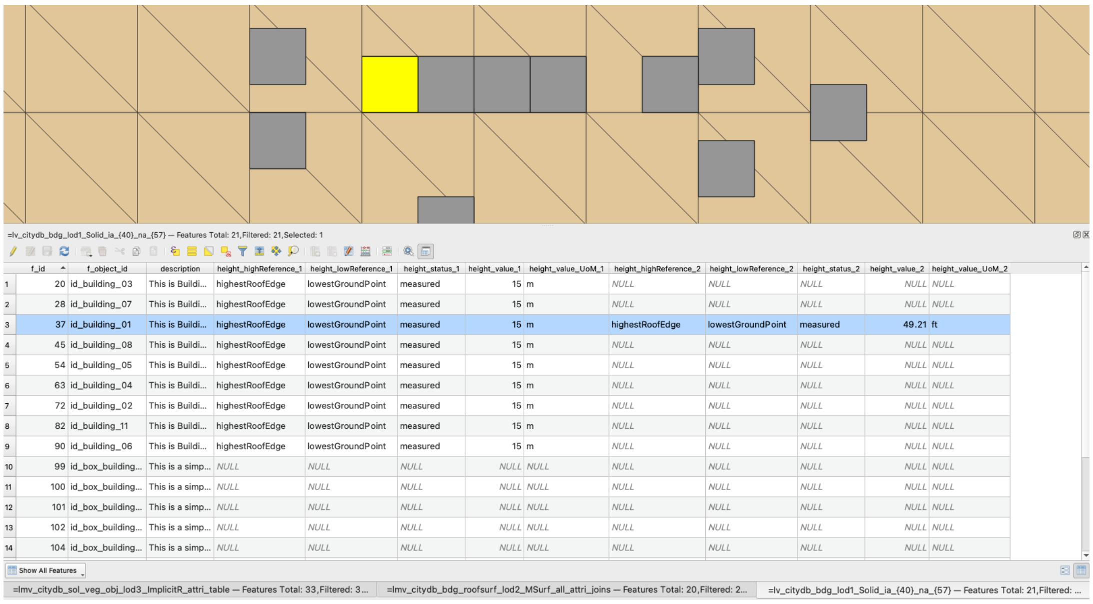
```
CREATE OR REPLACE VIEW ',usr_schema,'.',qi_layer_name,' AS
SELECT
    g.f_id,
    g.f_object_id,
    g.geom,
    -------------------------------------------------------------------
    -- Dynamically extended part
    a1.',qi_a1_col_name_2,',
    a2.',qi_a2_col_name_2,',
    a2.',qi_a2_col_name_3,'
    ...
    -------------------------------------------------------------------
FROM ',usr_schema,'.',qi_geometry_materialzied_view_name,' AS g
    -------------------------------------------------------------------
    -- Dynamically extended part
     LEFT JOIN ',usr_schema,'.',qi_attribute_view_name_1,' AS a1
        ON g.f_id = a1.f_id
     LEFT JOIN ',usr_schema,'.',qi_attribute_view_name_2,' AS a2
        ON g.f_id = a2.f_id;
     ...
     -------------------------------------------------------------------
```

Listing 6.1: Query template to creation layers using approach 1

Listing 6.2 shows two different modes to create layers of buildings in LoD1 Solid associated with two and all attributes using approach 1. Users can choose their desired feature geometry type and LoD for extracting the target feature geometry materialized view. The `create_geometry_view` function is invoked if the selected geometry view has not been created. For the attribute part, users can specify their desired attribute names in an array with the `is_all_attri` input parameter (the last two ones) set to `FALSE` to join the corresponding attribute views with geometries. Users can also choose to join all available attribute views regarding the feature geometries, which is achieved by leaving the `attri` (the attribute name array) input parameter to `NULL` with the `is_all_attri` input parameter set to `TRUE`. The generated layer of CASE 1 in Listing 6.2 is illustrated in Figure 6.1.

```sql
-- CASE 1 SELECTED ATTRIBUTES
-- Last three inputs: is_matview, is_all_attri, is_joins
SELECT qgis_pkg.create_layer('qgis_bstsai', 'citydb', 0, 901, 'lod1Solid',
        ARRAY['function','height'], FALSE, FALSE, TRUE);

-- CASE 2 ALL ATTRIBUTES
SELECT qgis_pkg.create_layer('qgis_bstsai', 'citydb', 0, 901, 'lod1Solid',
       NULL, FALSE, TRUE, TRUE);
```

Listing 6.2: Calling the `create_layer` function using approach 1



=lv_citydb_bdg_lod1_Solid_ia_{40}_na_{57} — Features Total: 21,Filtered: 21,Selected: 1

	f_id	f_object_id	description	height_highReference_1	height_lowReference_1	height_status_1	height_value_1	height_value_UoM_1	height_highReference_2	height_lowReference_2	height_status_2	height_value_2	height_value_UoM_2
1	20	id_building_03	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
2	28	id_building_07	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
3	37	id_building_01	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	highestRoofEdge	lowestGroundPoint	measured	49.21	ft
4	45	id_building_08	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
5	54	id_building_05	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
6	63	id_building_04	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
7	72	id_building_02	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
8	82	id_building_11	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
9	90	id_building_06	This is Buildi...	highestRoofEdge	lowestGroundPoint	measured	15	m	NULL	NULL	NULL	NULL	NULL
10	99	id_box_building...	This is a simp...	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11	100	id_box_building...	This is a simp...	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
12	101	id_box_building...	This is a simp...	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
13	102	id_box_building...	This is a simp...	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
14	104	id_box_building...	This is a simp...	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

=lmv_citydb_sol_veg_obj_lod3_ImplicitR_attri_table — Features Total: 33,Filtered: 3...

=lmv_citydb_bdg_roofsurf_lod2_MSurf_all_attri_joins — Features Total: 20,Filtered: 2...

=lv_citydb_bdg_lod1_Solid_ia_{40}_na_{57} — Features Total: 21,Filtered: ...

Figure 6.1.: Layer generated with approach 1 checked in QGIS
(Buildings in LoD1 Solid with "function" and "height" from the Alderaan dataset)

6.1.2. Approach 2- Joining Feature Geometry Materialized Views and Attribute Materialized Views

In approach 2, the target feature geometry materialized view is associated with the selected attribute materialized views for layer creation. If the materialized view of a selected attribute has not been created, the `create_layer` function calls the `create_attribute_view` function to create it first. Listing 6.3 provides the query template of approach 2. Similar to the query template of approach 1 (Listing 6.1), the `SELECT` and `FROM` clauses are dynamically extended based on the number of selected attributes. The main differences are the creation of indices on every column in the generated layer except those of `JSON` type. Note that only the attribute value columns in each attribute materialized view are extracted and indexed, with the enumeration of these columns starting from two, as the first column (`f_id`) is skipped.

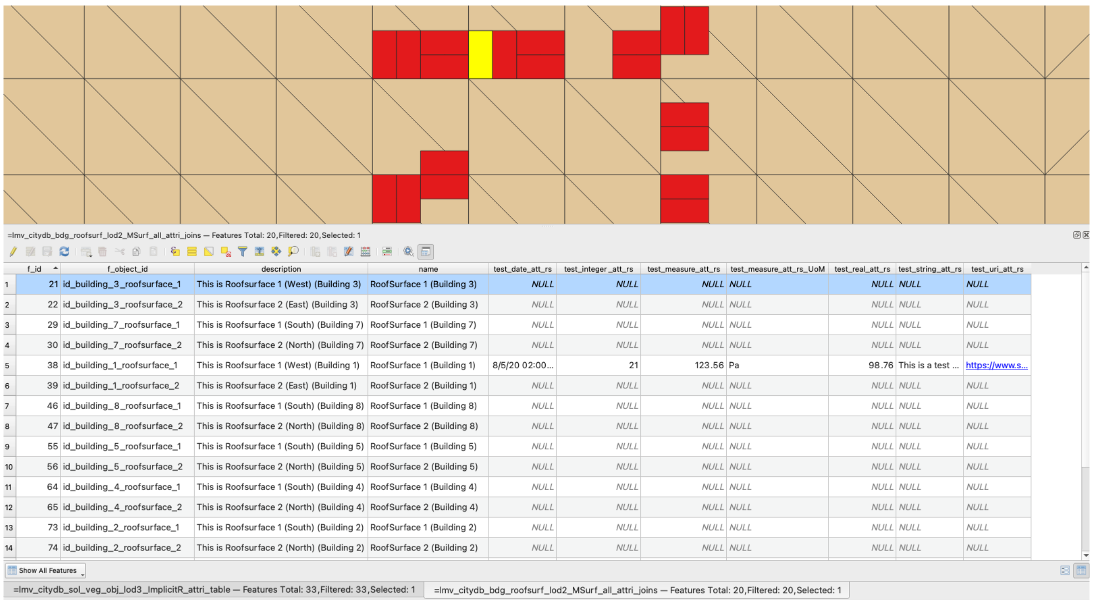
```
CREATE  MATERIALIZED  VIEW IF NOT EXISTS ',usr_schema,'.',qi_layer_name,' AS
SELECT
    g.f_id,
    g.f_object_id,
    g.geom,
    -------------------------------------------------------------------
    -- Dynamically extended part
    a1.',qi_a1_col_name_2,',
    a2.',qi_a2_col_name_2,',
    a2.',qi_a2_col_name_3,'
    ...
    -------------------------------------------------------------------
FROM ',usr_schema,'.',qi_geometry_materialzied_view_name,' AS g
    -------------------------------------------------------------------
    -- Dynamically extended part
    LEFT JOIN ',usr_schema,'.',qi_attri_materialized_view_name_1,' AS a1
        ON g.f_id = a1.f_id
    LEFT JOIN ',usr_schema,'.',qi_attri_materialized_view_name_2,' AS a2
        ON g.f_id = a2.f_id;
     ...
     -------------------------------------------------------------------
CREATE INDEX ',qi_layer_name,'_g_1_f_id_idx
    ON ',usr_schema,'.',qi_layer_name,' USING btree (f_id);
CREATE INDEX ',qi_layer_name,'_g_2_o_id_idx
    ON ',usr_schema,'.',qi_layer_name,' USING btree (f_object_id);
CREATE INDEX ',qi_layer_name,'_g_3_geom_spx
    ON ',usr_schema,'.',qi_layer_name,' USING gist (geom);
-------------------------------------------------------------------
-- Dynamically extended part
CREATE INDEX ',qi_layer_name,'_a1_2
    ON ',usr_schema,'.',qi_layer_name,' USING btree (',qi_a1_col_name_2,');
CREATE INDEX ',qi_layer_name,'_a2_2
    ON ',usr_schema,'.',qi_layer_name,' USING btree (',qi_a2_col_name_2,');
CREATE INDEX ',qi_layer_name,'_a2_3
    ON ',usr_schema,'.',qi_layer_name,' USING btree (',qi_a2_col_name_3,');
...
-------------------------------------------------------------------
```

Listing 6.3: Query template to creation layers using approach 2

Listing 6.4 shows two different modes to create layers of building roofs in LoD2 Multi-Surface associated with two and all attributes using approach 2. The function input specifications are similar to Listing 6.2, with the difference being that `is_matview` is set to `TRUE` for creating individual attributes and storing the generated layer in materialized views. The generated layer of CASE 2 in Listing 6.4 is illustrated in Figure 6.2.

```sql
-- CASE 1 SELECTED ATTRIBUTES
-- Last three inputs: is_matview, is_all_attri, is_joins
SELECT qgis_pkg.create_layer('qgis_bstsai', 'citydb', 901, 712,
    'lod2MultiSurface', ARRAY['name','description'], TRUE, FALSE, TRUE);

-- CASE 2 ALL ATTRIBUTES
SELECT qgis_pkg.create_layer('qgis_bstsai', 'citydb', 901, 712,
    'lod2MultiSurface', NULL, TRUE, TRUE, TRUE);
```

Listing 6.4: Calling the `create_layer` function using approach 2



=lmv_citydb_bdg_roofsurf_lod2_MSurf_all_attri_joins — Features Total: 20,Filtered: 20,Selected: 1

	f_id	f_object_id	description	name	test_date_att_rs	test_integer_att_rs	test_measure_att_rs	test_measure_att_rs_UoM	test_real_att_rs	test_string_att_rs	test_uri_att_rs
1	21	id_building_3_roofsurface_1	This is Roofsurface 1 (West) (Building 3)	RoofSurface 1 (Building 3)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
2	22	id_building_3_roofsurface_2	This is Roofsurface 2 (East) (Building 3)	RoofSurface 2 (Building 3)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	29	id_building_7_roofsurface_1	This is Roofsurface 1 (South) (Building 7)	RoofSurface 1 (Building 7)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	30	id_building_7_roofsurface_2	This is Roofsurface 2 (North) (Building 7)	RoofSurface 2 (Building 7)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	38	id_building_1_roofsurface_1	This is Roofsurface 1 (West) (Building 1)	RoofSurface 1 (Building 1)	8/5/20 02:00...	21	123.56	Pa	98.76	This is a test ...	https://www.s...
6	39	id_building_1_roofsurface_2	This is Roofsurface 2 (East) (Building 1)	RoofSurface 2 (Building 1)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
7	46	id_building_8_roofsurface_1	This is Roofsurface 1 (South) (Building 8)	RoofSurface 1 (Building 8)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
8	47	id_building_8_roofsurface_2	This is Roofsurface 2 (North) (Building 8)	RoofSurface 2 (Building 8)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
9	55	id_building_5_roofsurface_1	This is Roofsurface 1 (South) (Building 5)	RoofSurface 1 (Building 5)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
10	56	id_building_5_roofsurface_2	This is Roofsurface 2 (North) (Building 5)	RoofSurface 2 (Building 5)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11	64	id_building_4_roofsurface_1	This is Roofsurface 1 (South) (Building 4)	RoofSurface 1 (Building 4)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
12	65	id_building_4_roofsurface_2	This is Roofsurface 2 (North) (Building 4)	RoofSurface 2 (Building 4)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
13	73	id_building_2_roofsurface_1	This is Roofsurface 1 (South) (Building 2)	RoofSurface 1 (Building 2)	NULL	NULL	NULL	NULL	NULL	NULL	NULL
14	74	id_building_2_roofsurface_2	This is Roofsurface 2 (North) (Building 2)	RoofSurface 2 (Building 2)	NULL	NULL	NULL	NULL	NULL	NULL	NULL

=lmv_citydb_sol_veg_obj_lod3_ImplicitR_attri_table — Features Total: 33,Filtered: 33,Selected: 1

=lmv_citydb_bdg_roofsurf_lod2_MSurf_all_attri_joins — Features Total: 20,Filtered: 20,Selected: 1

Figure 6.2.: Layer generated with approach 2 checked in QGIS
(Building roofs in LoD2 Multi-surface with all attributes from the Alderaan dataset)

6.1.3. Approach 3- Joining Feature Geometry Materialized Views and Attribute Table Materialized Views

In approach 3, the selected attributes are first gathered into an integrated attribute table and stored in materialized views. This attribute table is then associated with the target feature geometry materialized view for layer creation. If the attribute table of the selected attributes has not been created, the `create_layer` function calls the `create_attris_table_view` function to generate it by dynamically filling the query template shown in Listing 6.5. The attribute table query template uses the `FULL JOIN` function to gather the selected attributes by joining them on their `f_ids` union. The `SELECT` and `FROM` clauses are dynamically extended based on the candidate attribute names and columns, and indices are created on every attribute value column correspondingly.

The naming of the integrated attribute table consists of indicator labels, such as "_av_" for "attribute view" and "_amv_" for "attribute materialized view", along with the `cdb_schema` name, class alias, a "_g_" label followed by the `geometry_id` corresponding to the selected feature geometry, and an ”attribute” label. For instance, if users choose to create a layer of buildings represented in LoD1 Solid (geometry_id = 14 in the geometry metadata table) and join it with the desired attributes, the specified attributes are then stored in the integrated attribute table named _amv_citydb_Building_g_14_attributes.

```
CREATE  MATERIALIZED  VIEW IF NOT EXISTS ',usr_schema,'.',qi_attri_table,' AS
----------------------------------------------------------------------
-- Dynamically extended part
SELECT
    COALESCE(a1.f_id, a2.f_id, ...) AS f_id,
    a1.',qi_a1_col_name_2,',
    a2.',qi_a2_col_name_2,',
    a2.',qi_a2_col_name_3,',
    ...
FROM (',ql_collect_flatten_attri_1_SQL,') AS a1
    FULL JOIN (',ql_collect_flatten_attri_2_SQL,') AS a2
        ON COALESCE(a1.f_id) = a2.f_id
    FULL JOIN (',ql_collect_flatten_attri_3_SQL,') AS a3
        ON COALESCE(a1.f_id, a2.f_id) = a3.f_id
    ...
CREATE INDEX ',qi_attri_table,'_a1_1
    ON ',usr_schema,'.',qi_attri_table_name,' USING btree (f_id);
CREATE INDEX ',qi_attri_table_name,'_a1_2
    ON ',usr_schema,'.',qi_attri_table,' USING btree (',qi_a1_col_name_2,');
CREATE INDEX ',qi_attri_table_name,'_a2_2
    ON ',usr_schema,'.',qi_attri_table,' USING btree (',qi_a2_col_name_2,');
CREATE INDEX ',qi_attri_table_name,'_a2_3
    ON ',usr_schema,'.',qi_attri_table,' USING btree (',qi_a2_col_name_3,');
...
----------------------------------------------------------------------
```

Listing 6.5: Query template to create integrated attribute table view for approach 3

Listing 6.6 gives the query template of approach 3. Compared to approaches 1 and 2, the prominent difference is that the LEFT JOIN operation for the attributes is reduced to a single instance, as all selected attributes are stored in the integrated attribute table and then associated with feature geometries. Only the SELECT clause and attribute index creation query are dynamically extended based on the selected attributes.

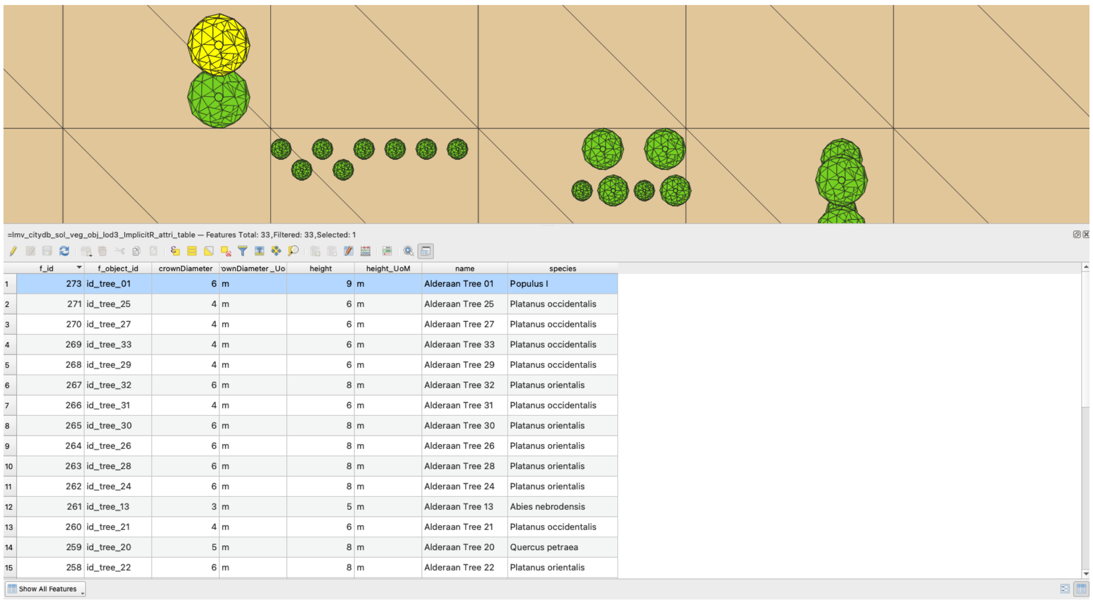
Listing 6.7 shows two different modes to create layers of trees in LoD3 Implicit-Representation associated with two and all attributes using approach 3. Similar to Listing 6.2 and Listing 6.4, the main differences in the function input specifications are that the last input is_joined is omitted for the default FALSE value, and is_matview is set to TRUE for integrating all selected attributes into the attribute table and storing the generated layer in materialized views. The generated layer of CASE 2 in Listing 6.7 is illustrated in Figure 6.3.

```
CREATE  MATERIALIZED  VIEW IF NOT EXISTS ',usr_schema,'.',qi_layer_name,' AS
SELECT
    g.f_id,
    g.f_object_id,
    g.geom,
    -------------------------------------------------------------------
    -- Dynamically extended part
    -- The first column a.f_id is not selected
    a.',qi_a_col_name_2,',
    a.',qi_a_col_name_3,',
    a.',qi_a_col_name_4,'
    ...
    -------------------------------------------------------------------
FROM ',usr_schema,'.',qi_geometry_materialzied_view_name,' AS g
    LEFT JOIN ',usr_schema,'.',qi_attri_table,' AS a ON g.f_id = a.f_id

CREATE INDEX ',qi_layer_name,'_g_1_f_id_idx
    ON ',usr_schema,'.',qi_layer_name,' USING btree (f_id);
CREATE INDEX ',qi_layer_name,'_g_2_o_id_idx
    ON ',usr_schema,'.',qi_layer_name,' USING btree (f_object_id);
CREATE INDEX ',qi_layer_name,'_g_3_geom_spx
    ON ',usr_schema,'.',qi_layer_name,' USING gist (geom);
-----------------------------------------------------------------------
-- Dynamically extended part
CREATE INDEX ',qi_layer_name,'_a_2
    ON ',usr_schema,'.',qi_layer_name,' USING btree (',qi_a_col_name_2,');
CREATE INDEX ',qi_layer_name,'_a_2
    ON ',usr_schema,'.',qi_layer_name,' USING btree (',qi_a_col_name_2,');
CREATE INDEX ',qi_layer_name,'_a_3
    ON ',usr_schema,'.',qi_layer_name,' USING btree (',qi_a_col_name_3,');
...
-----------------------------------------------------------------------
```

Listing 6.6: Query template to creation layers using approach 3

```
-- CASE 1 SELECTED ATTRIBUTES
-- Last input is_joins is omitted for the default FALSE
SELECT qgis_pkg.create_layer('qgis_bstsai', 'citydb', 0, 1301,
    'lod3ImplicitRepresentation', ARRAY['name','species'], TRUE, FALSE);

-- CASE 2 ALL ATTRIBUTES
SELECT qgis_pkg.create_layer('qgis_bstsai', 'citydb', 0, 1301,
    'lod3ImplicitRepresentation', NULL, TRUE, TRUE);
```

Listing 6.7: Calling the `create_layer` function using approach 3



=lmv_citydb_sol_veg_obj_lod3_ImplicitR_attri_table — Features Total: 33,Filtered: 33,Selected: 1

	f_id	f_object_id	crownDiameter	ownDiameter_Uo	height	height_UoM	name	species
1	273	id_tree_01	6	m	9	m	Alderaan Tree 01	Populus l
2	271	id_tree_25	4	m	6	m	Alderaan Tree 25	Platanus occidentalis
3	270	id_tree_27	4	m	6	m	Alderaan Tree 27	Platanus occidentalis
4	269	id_tree_33	4	m	6	m	Alderaan Tree 33	Platanus occidentalis
5	268	id_tree_29	4	m	6	m	Alderaan Tree 29	Platanus occidentalis
6	267	id_tree_32	6	m	8	m	Alderaan Tree 32	Platanus orientalis
7	266	id_tree_31	4	m	6	m	Alderaan Tree 31	Platanus occidentalis
8	265	id_tree_30	6	m	8	m	Alderaan Tree 30	Platanus orientalis
9	264	id_tree_26	6	m	8	m	Alderaan Tree 26	Platanus orientalis
10	263	id_tree_28	6	m	8	m	Alderaan Tree 28	Platanus orientalis
11	262	id_tree_24	6	m	8	m	Alderaan Tree 24	Platanus orientalis
12	261	id_tree_13	3	m	5	m	Alderaan Tree 13	Abies nebrodensis
13	260	id_tree_21	4	m	6	m	Alderaan Tree 21	Platanus occidentalis
14	259	id_tree_20	5	m	8	m	Alderaan Tree 20	Quercus petraea
15	258	id_tree_22	6	m	8	m	Alderaan Tree 22	Platanus orientalis

Figure 6.3.: Layer generated with approach 3 checked in QGIS
(Trees in LoD3 implicit representation with all attributes from the Alderaan dataset)

6.2. Layer Query Performance Test

For determining the approach of GIS layer creation, query performance assessments of the generated GIS layers are conducted on both the Rijsen-Holten and Vienna datasets to evaluate the query time efficiency of the different layer creation approaches.

Table 6.1 and Table 6.2 present the results of the GIS layer query performance tests. Although approaches 2 and 3 consume storage space and take longer generation time due to the indexing of materialized views, they significantly outperform approach 1 regarding layer query speed. Notably, approach 3 provides flexible layer management by reducing the number of individual attribute views and the `LEFT JOIN` operations between feature geometries and selected attributes. The selected attributes are gathered and stored in an integrated attribute table, which is dropped and created each time users specify the target feature geometries. It offers an up-to-date attributes table for user reference.

Additionally, fewer joins could be a better choice because each `LEFT JOIN` adds computational overhead, increasing the complexity and execution time of the query. By minimizing the number of joins, approach 3 could not only speed up query processing but also reduce the likelihood of potential bottlenecks, making it the optimal approach for fast GIS layer querying.

Dataset	Parent Class (objectclass_id)	Class (objectclass_id)	LoD	Geometry type and Attributes	Query Conditions	Approach	Layer Query Time	Layer Generate Time	Number of feature
Rijsen-Holten	-	Building (901)	0	Multi-Surface All available attributes	• date of Construction >= '1990-01-01' • storeys Above Ground >= 2 • height_value >= 6	1- Views (Joins)	>1 hour	00:23.7	3,936
						2- Materialized Views (Joins)	00:00.4	00:35.3	
						3- Materialized View (Table Join)	00:00.1	00:38.4	
	Building (901)	RoofSurface (712)	2	Multi-Surface All available attributes	• direction = ('SE','S','SW') • lod2_area >= 30	1- Views (Joins)	00:07.5	00:25.6	21,021
						2- Materialized Views (Joins)	00:00.3	00:47.8	
						3- Materialized View (Table Join)	00:00.4	01:02.6	
	-	SolitaryVegetation Object (1301)	3	Implicit Representation All available attributes	• crown Diameter >= 10 • height >= 5	1- Views (Joins)	00:02.4	07:37.5	4,638
						2- Materialized Views (Joins)	00:01.3	00:15.2	
						3- Materialized View (Table Join)	00:00.9	00:11.6	

time unit: mm:ss.s

Table 6.1.: GIS layers query performance test (Rijsen-Holten dataset)

Dataset	Parent Class (objectclass_id)	Class (objectclass_id)	LoD	Geometry type and Attributes	Query Conditions	Approach	Layer Query Time	Layer Generate Time	Number of feature
Vienna	BuildingPart (902)	RoofSurface (712)	2	Multi-Surface All available attributes	• 45<= normal <= 90 • 90 <= orientation <= 270 • area >= 30	1- Views (Joins)	01:56.6	02:20.8	110,395
						2- Materialized Views (Joins)	00:00.7	07:28.5	
						3- Materialized View (Table Join)	00:00.3	07:13.6	
	-	SolitaryVegetation Object (1301)	2	Implicit Representation All available attributes	• crown Diameter >= 8 • height >= 5 • trunk_Diameter >= 0.3 • 1900 <= year_of_plantation <= 2010 • species LIKE 'Acer%'	1- Views (Joins)	00:23.6	00:35.1	3,341
						2- Materialized Views (Joins)	00:01.0	01:45.6	
						3- Materialized View (Table Join)	00:00.4	01:45.5	

time unit: mm:ss.s

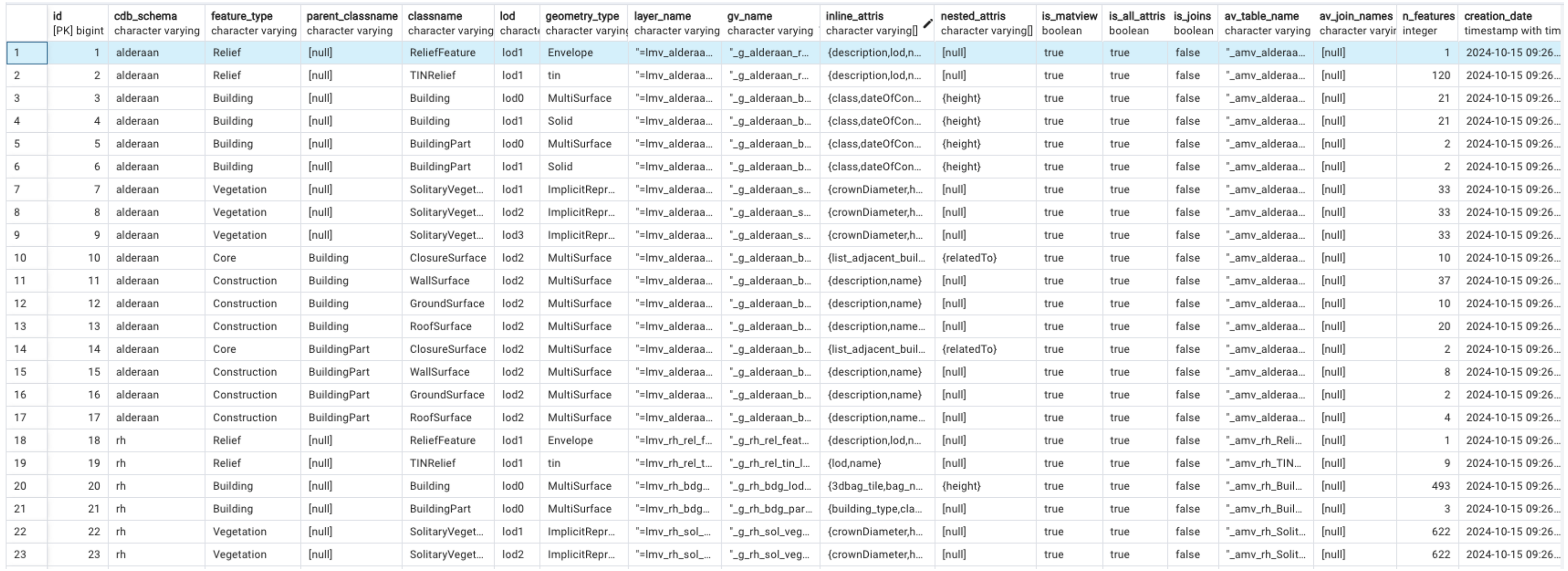
Table 6.2.: GIS layers query performance test (Vienna dataset)

6.3. Layer Metadata Table Design

As indicated in Section 4.4, the design of the LAYER_METADATA table is shown in the Table 6.3. The LAYER_METADATA table created under the `cdb_schema` stores the metadata of the generated GIS layers, which can be used for layer management. The populated result after GIS layer generation is shown in Figure 6.4.

Column Name	PostgreSQL Type	Values
id	bigint[PK]	Primary key
cdb_schema	varchar	Target database schema
feature_type	varchar	CityGML module name
parent_objectclass_id	integer	Parent class ID of boundary features. NULL as default
parent_classname	varchar	Name of the parent class. NULL as default
objectclass_id	integer	class ID of the features
classname	varchar	Name of the class
lod	varchar	LoD value
geometry_type	varchar	Solid, MultiSurface, etc
layer_name	varchar	Generated layer name
gv_name	varchar	Selected geometry view name
inline_attris	varchar[]	Selected inline attributes
nested_attris	varchar[]	Selected nested attributes
is_matview	boolean	Identifier of layer materialized view creation
is_all_attris	boolean	Identifier of all existing attributes selection for the target class
is_joins	boolean	Identifier of layer creation approach. False as default. True for using approaches 1 or 2
av_table_name	varchar	joined attribute table view name when approach 3 is applied. Null for using approach 1 or 2
av_join_names	varchar	joined attribute view names when approach 1 or 2 is applied. Null for using approach 3
n_features	integer	number of features in the generated layer
creation_time	timestamp(3)	layer creation time

Table 6.3.: LAYER_METADATA table design



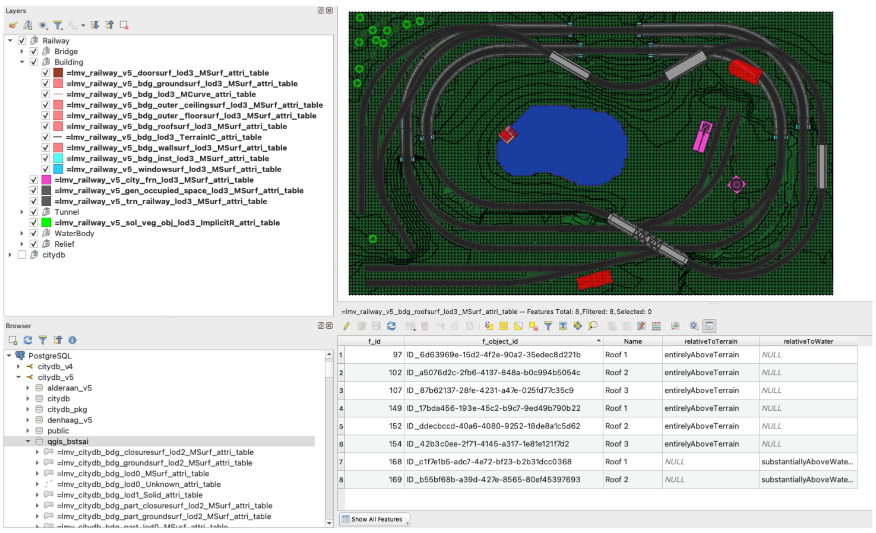
	id	cdb_schema	feature_type	parent_classname	classname	lod	geometry_type	layer_name	gv_name	inline_attris	nested_attris	is_matview	is_all_attris	is_joins	av_table_name	av_join_names	n_features	creation_date
	[PK] bigint	character varying	character varying	character varying	character varying	characte	character varyin	character varying	character varying	character varying[]	character varying[]	boolean	boolean	boolean	character varying	character varyir	integer	timestamp with tim
1	1	alderaan	Relief	[null]	ReliefFeature	lod1	Envelope	"=lmv_alderaa...	"_g_alderaan_r...	{description,lod,n...	[null]	true	true	false	"_amv_alderaa...	[null]	1	2024-10-15 09:26...
2	2	alderaan	Relief	[null]	TINRelief	lod1	tin	"=lmv_alderaa...	"_g_alderaan_r...	{description,lod,n...	[null]	true	true	false	"_amv_alderaa...	[null]	120	2024-10-15 09:26...
3	3	alderaan	Building	[null]	Building	lod0	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{class,dateOfCon...	{height}	true	true	false	"_amv_alderaa...	[null]	21	2024-10-15 09:26...
4	4	alderaan	Building	[null]	Building	lod1	Solid	"=lmv_alderaa...	"_g_alderaan_b...	{class,dateOfCon...	{height}	true	true	false	"_amv_alderaa...	[null]	21	2024-10-15 09:26...
5	5	alderaan	Building	[null]	BuildingPart	lod0	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{class,dateOfCon...	{height}	true	true	false	"_amv_alderaa...	[null]	2	2024-10-15 09:26...
6	6	alderaan	Building	[null]	BuildingPart	lod1	Solid	"=lmv_alderaa...	"_g_alderaan_b...	{class,dateOfCon...	{height}	true	true	false	"_amv_alderaa...	[null]	2	2024-10-15 09:26...
7	7	alderaan	Vegetation	[null]	SolitaryVeget...	lod1	ImplicitRepr...	"=lmv_alderaa...	"_g_alderaan_s...	{crownDiameter,h...	[null]	true	true	false	"_amv_alderaa...	[null]	33	2024-10-15 09:26...
8	8	alderaan	Vegetation	[null]	SolitaryVeget...	lod2	ImplicitRepr...	"=lmv_alderaa...	"_g_alderaan_s...	{crownDiameter,h...	[null]	true	true	false	"_amv_alderaa...	[null]	33	2024-10-15 09:26...
9	9	alderaan	Vegetation	[null]	SolitaryVeget...	lod3	ImplicitRepr...	"=lmv_alderaa...	"_g_alderaan_s...	{crownDiameter,h...	[null]	true	true	false	"_amv_alderaa...	[null]	33	2024-10-15 09:26...
10	10	alderaan	Core	Building	ClosureSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{list_adjacent_buil...	{relatedTo}	true	true	false	"_amv_alderaa...	[null]	10	2024-10-15 09:26...
11	11	alderaan	Construction	Building	WallSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{description,name}	[null]	true	true	false	"_amv_alderaa...	[null]	37	2024-10-15 09:26...
12	12	alderaan	Construction	Building	GroundSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{description,name}	[null]	true	true	false	"_amv_alderaa...	[null]	10	2024-10-15 09:26...
13	13	alderaan	Construction	Building	RoofSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{description,name...	[null]	true	true	false	"_amv_alderaa...	[null]	20	2024-10-15 09:26...
14	14	alderaan	Core	BuildingPart	ClosureSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{list_adjacent_buil...	{relatedTo}	true	true	false	"_amv_alderaa...	[null]	2	2024-10-15 09:26...
15	15	alderaan	Construction	BuildingPart	WallSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{description,name}	[null]	true	true	false	"_amv_alderaa...	[null]	8	2024-10-15 09:26...
16	16	alderaan	Construction	BuildingPart	GroundSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{description,name}	[null]	true	true	false	"_amv_alderaa...	[null]	2	2024-10-15 09:26...
17	17	alderaan	Construction	BuildingPart	RoofSurface	lod2	MultiSurface	"=lmv_alderaa...	"_g_alderaan_b...	{description,name...	[null]	true	true	false	"_amv_alderaa...	[null]	4	2024-10-15 09:26...
18	18	rh	Relief	[null]	ReliefFeature	lod1	Envelope	"=lmv_rh_rel_f...	"_g_rh_rel_feat...	{description,lod,n...	[null]	true	true	false	"_amv_rh_Reli...	[null]	1	2024-10-15 09:26...
19	19	rh	Relief	[null]	TINRelief	lod1	tin	"=lmv_rh_rel_t...	"_g_rh_rel_tin_l...	{lod,name}	[null]	true	true	false	"_amv_rh_TIN...	[null]	9	2024-10-15 09:26...
20	20	rh	Building	[null]	Building	lod0	MultiSurface	"=lmv_rh_bdg...	"_g_rh_bdg_lod...	{3dbag_tile,bag_n...	{height}	true	true	false	"_amv_rh_Buil...	[null]	493	2024-10-15 09:26...
21	21	rh	Building	[null]	BuildingPart	lod0	MultiSurface	"=lmv_rh_bdg...	"_g_rh_bdg_par...	{building_type,cla...	[null]	true	true	false	"_amv_rh_Buil...	[null]	3	2024-10-15 09:26...
22	22	rh	Vegetation	[null]	SolitaryVeget...	lod1	ImplicitRepr...	"=lmv_rh_sol_...	"_g_rh_sol_veg...	{crownDiameter,h...	[null]	true	true	false	"_amv_rh_Solit...	[null]	622	2024-10-15 09:26...
23	23	rh	Vegetation	[null]	SolitaryVeget...	lod2	ImplicitRepr...	"=lmv_rh_sol_...	"_g_rh_sol_veg...	{crownDiameter,h...	[null]	true	true	false	"_amv_rh_Solit...	[null]	622	2024-10-15 09:26...

Figure 6.4.: Populated LAYER_METADATA table example

6.4. Generated Layer Results

This section presents the visualised results in QGIS of the GIS layers generated by approach 3 from different datasets listed in Table 5.1.

- **Railway (CityGML2.0)**: This dataset is used to test the layer generation of classes across various CityGML modules, including "Bridge", "Building", "CityFurniture", "Generics", "Transportation", "Tunnel", "Vegetation", "WaterBody", and "Relief". Figure 6.5 and Figure 6.6 show the generated layers viewed in 2D and 3D using QGIS.



```
▾ ✓ Railway
  ▸ ✓ Bridge
  ▾ ✓ Building
      ✓ =lmv_railway_v5_doorsurf_lod3_MSurf_attri_table
      ✓ =lmv_railway_v5_bdg_groundsurf_lod3_MSurf_attri_table
      ✓ =lmv_railway_v5_bdg_lod3_MCurve_attri_table
      ✓ =lmv_railway_v5_bdg_outer_ceilingsurf_lod3_MSurf_attri_table
      ✓ =lmv_railway_v5_bdg_outer_floorsurf_lod3_MSurf_attri_table
      ✓ =lmv_railway_v5_bdg_roofsurf_lod3_MSurf_attri_table
      ✓ =lmv_railway_v5_bdg_lod3_TerrainIC_attri_table
      ✓ =lmv_railway_v5_bdg_wallsurf_lod3_MSurf_attri_table
      ✓ =lmv_railway_v5_bdg_inst_lod3_MSurf_attri_table
      ✓ =lmv_railway_v5_windowsurf_lod3_MSurf_attri_table
    ✓ =lmv_railway_v5_city_frn_lod3_MSurf_attri_table
    ✓ =lmv_railway_v5_gen_occupied_space_lod3_MSurf_attri_table
    ✓ =lmv_railway_v5_trn_railway_lod3_MSurf_attri_table
  ▸ ✓ Tunnel
    ✓ =lmv_railway_v5_sol_veg_obj_lod3_ImplicitR_attri_table
  ▸ ✓ WaterBody
  ▸ ✓ Relief
▸   citydb
```

```
▾ PostgreSQL
  ▸ citydb_v4
  ▾ citydb_v5
    ▸ alderaan_v5
    ▸ citydb
    ▸ citydb_pkg
    ▸ denhaag_v5
    ▸ public
    ▾ qgis_bstsai
      ▸ =lmv_citydb_bdg_closuresurf_lod2_MSurf_attri_table
      ▸ =lmv_citydb_bdg_groundsurf_lod2_MSurf_attri_table
      ▸ =lmv_citydb_bdg_lod0_MSurf_attri_table
      ▸ =lmv_citydb_bdg_lod0_Unknown_attri_table
      ▸ =lmv_citydb_bdg_lod1_Solid_attri_table
      ▸ =lmv_citydb_bdg_part_closuresurf_lod2_MSurf_attri_table
      ▸ =lmv_citydb_bdg_part_groundsurf_lod2_MSurf_attri_table
```

=lmv_railway_v5_bdg_roofsurf_lod3_MSurf_attri_table — Features Total: 8,Filtered: 8,Selected: 0

	f_id	f_object_id	Name	relativeToTerrain	relativeToWater
1	97	ID_6d63969e-15d2-4f2e-90a2-35edec8d221b	Roof 1	entirelyAboveTerrain	NULL
2	102	ID_a5076d2c-2fb6-4137-848a-b0c994b5054c	Roof 2	entirelyAboveTerrain	NULL
3	107	ID_87b62137-28fe-4231-a47e-025fd77c35c9	Roof 3	entirelyAboveTerrain	NULL
4	149	ID_17bda456-193e-45c2-b9c7-9ed49b790b22	Roof 1	entirelyAboveTerrain	NULL
5	152	ID_ddecbccd-40a6-4080-9252-18de8a1c5d62	Roof 2	entirelyAboveTerrain	NULL
6	154	ID_42b3c0ee-2f71-4145-a317-1e81e121f7d2	Roof 3	entirelyAboveTerrain	NULL
7	168	ID_c1f7e1b5-adc7-4e72-bf23-b2b31dcc0368	Roof 1	NULL	substantiallyAboveWate...
8	169	ID_b55bf68b-a39d-427e-8565-80ef45397693	Roof 2	NULL	substantiallyAboveWate...

Figure 6.5.: Generated layers of the Railway dataset checked in QGIS (2D)

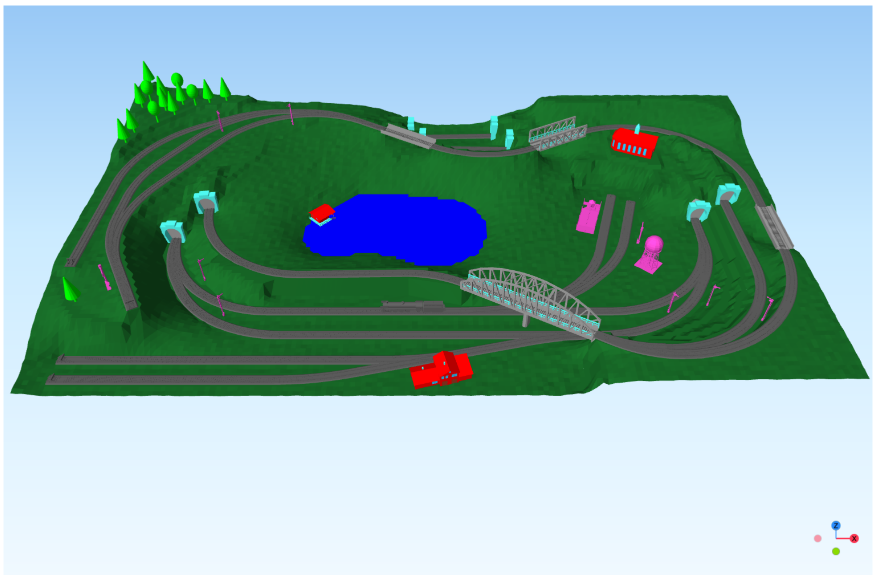
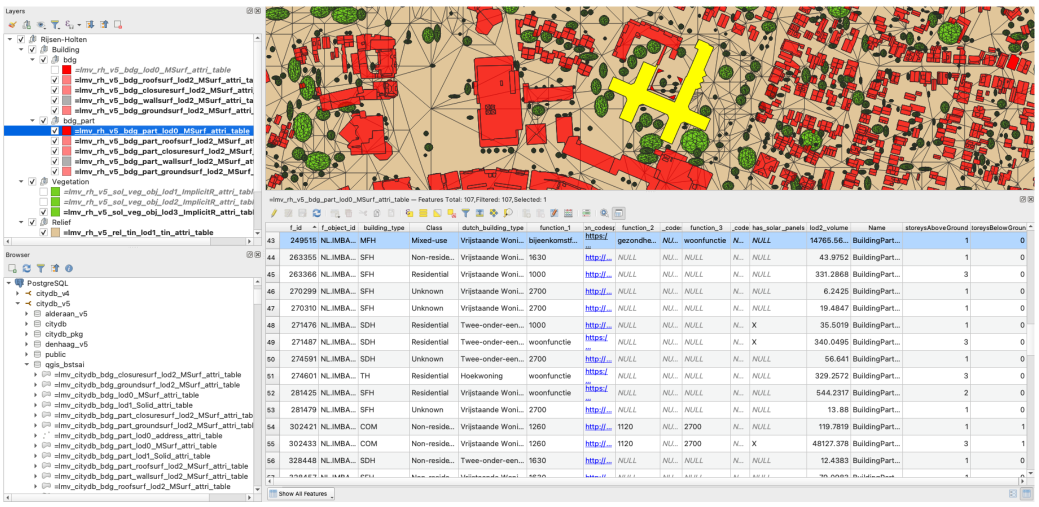


Figure 6.6.: Generated layers of the Railway dataset checked in QGIS (3D)

- **Rijsen-Holten (CityGML2.0)**: This real-world dataset, rich in attributes, is used to test layer generation for common modules such as "Building", "Vegetation", and "Relief". Figure 6.7 and Figure 6.8 demonstrate the zoomed-in, full-database extent of layer generation for all existing feature geometries and their attributes. In these figures, building roofs and walls are represented in LoD2 Multi-Surface, coloured in red and grey, respectively. Trees are shown in implicit representation and coloured in green, while the relief triangular irregular network (TIN) in LoD1 is coloured in yellow. Figure 6.9 and Figure 6.10 present the results of layer generation with extent selection.



- ✓ Rijsen-Holten
  - ✓ Building
    - ✓ bdg
      - 🟥 *=lmv_rh_v5_bdg_lod0_MSurf_attri_table*
      - ✓ 🟥 =lmv_rh_v5_bdg_roofsurf_lod2_MSurf_attri_ta
      - ✓ 🟥 =lmv_rh_v5_bdg_closuresurf_lod2_MSurf_attri_
      - ✓ ⬜ =lmv_rh_v5_bdg_wallsurf_lod2_MSurf_attri_ta
      - ✓ 🟥 =lmv_rh_v5_bdg_groundsurf_lod2_MSurf_attri_
    - ✓ bdg_part
      - ✓ 🟥 =lmv_rh_v5_bdg_part_lod0_MSurf_attri_table
      - ✓ 🟥 =lmv_rh_v5_bdg_part_roofsurf_lod2_MSurf_att
      - ✓ 🟥 =lmv_rh_v5_bdg_part_closuresurf_lod2_MSurf_
      - ✓ ⬜ =lmv_rh_v5_bdg_part_wallsurf_lod2_MSurf_att
      - ✓ 🟥 =lmv_rh_v5_bdg_part_groundsurf_lod2_MSurf_
  - ✓ Vegetation
    - 🟩 *=lmv_rh_v5_sol_veg_obj_lod1_ImplicitR_attri_tabl*
    - 🟩 *=lmv_rh_v5_sol_veg_obj_lod2_ImplicitR_attri_tabl*
    - ✓ 🟩 =lmv_rh_v5_sol_veg_obj_lod3_ImplicitR_attri_tabl
  - ✓ Relief
    - ✓ 🟨 =lmv_rh_v5_rel_tin_lod1_tin_attri_table

- PostgreSQL
  - citydb_v4
  - citydb_v5
    - alderaan_v5
    - citydb
    - citydb_pkg
    - denhaag_v5
    - public
    - qgis_bstsai
      - =lmv_citydb_bdg_closuresurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_groundsurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_lod0_MSurf_attri_table
      - =lmv_citydb_bdg_lod1_Solid_attri_table
      - =lmv_citydb_bdg_part_closuresurf_lod2_MSurf_attri_tab
      - =lmv_citydb_bdg_part_groundsurf_lod2_MSurf_attri_tabl
      - =lmv_citydb_bdg_part_lod0_address_attri_table
      - =lmv_citydb_bdg_part_lod0_MSurf_attri_table
      - =lmv_citydb_bdg_part_lod1_Solid_attri_table
      - =lmv_citydb_bdg_part_roofsurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_part_wallsurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_roofsurf_lod2_MSurf_attri_table

=lmv_rh_v5_bdg_part_lod0_MSurf_attri_table — Features Total: 107,Filtered: 107,Selected: 1

	f_id	f_object_id	building_type	Class	dutch_building_type	function_1	on_codesp	function_2	_code:	function_3	_code	has_solar_panels	lod2_volume	Name	storeysAboveGround	toreysBelowGroun
43	249515	NL.IMBA...	MFH	Mixed-use	Vrijstaande Woni...	bijeenkomstf...	https:/ ...	gezondhe...	NU...	woonfunctie	N...	NULL	14765.56...	BuildingPart...	1	0
44	263355	NL.IMBA...	SFH	Non-reside...	Vrijstaande Woni...	1630	http://...	NULL	NU...	NULL	N...	NULL	43.9752	BuildingPart...	1	0
45	263366	NL.IMBA...	SFH	Residential	Vrijstaande Woni...	1000	http://...	NULL	NU...	NULL	N...	NULL	331.2868	BuildingPart...	3	0
46	270299	NL.IMBA...	SFH	Unknown	Vrijstaande Woni...	2700	http://...	NULL	NU...	NULL	N...	NULL	6.2425	BuildingPart...	1	0
47	270310	NL.IMBA...	SFH	Unknown	Vrijstaande Woni...	2700	http://...	NULL	NU...	NULL	N...	NULL	19.4847	BuildingPart...	1	0
48	271476	NL.IMBA...	SDH	Residential	Twee-onder-een...	1000	http://...	NULL	NU...	NULL	N...	X	35.5019	BuildingPart...	1	0
49	271487	NL.IMBA...	SDH	Residential	Twee-onder-een...	woonfunctie	https:/ ...	NULL	NU...	NULL	N...	X	340.0495	BuildingPart...	3	0
50	274591	NL.IMBA...	SDH	Unknown	Twee-onder-een...	2700	http://...	NULL	NU...	NULL	N...	NULL	56.641	BuildingPart...	1	0
51	274601	NL.IMBA...	TH	Residential	Hoekwoning	woonfunctie	https:/ ...	NULL	NU...	NULL	N...	NULL	329.2572	BuildingPart...	3	0
52	281425	NL.IMBA...	SFH	Residential	Vrijstaande Woni...	woonfunctie	https:/ ...	NULL	NU...	NULL	N...	NULL	544.2317	BuildingPart...	2	0
53	281479	NL.IMBA...	SFH	Unknown	Vrijstaande Woni...	2700	http://...	NULL	NU...	NULL	N...	NULL	13.88	BuildingPart...	1	0
54	302421	NL.IMBA...	COM	Non-reside...	Vrijstaande Woni...	1260	http://...	1120	NU...	2700	N...	NULL	119.7819	BuildingPart...	1	1
55	302433	NL.IMBA...	COM	Non-reside...	Vrijstaande Woni...	1260	http://...	1120	NU...	2700	N...	X	48127.378	BuildingPart...	3	1
56	328448	NL.IMBA...	SDH	Non-reside...	Twee-onder-een...	1630	http://...	NULL	NU...	NULL	N...	NULL	12.4383	BuildingPart...	1	0

Figure 6.7.: Generated layers of the Rijsen-Holten dataset (full extent) checked in QGIS (2D) (Buildings layer in LoD0 Multi-Surface selected)

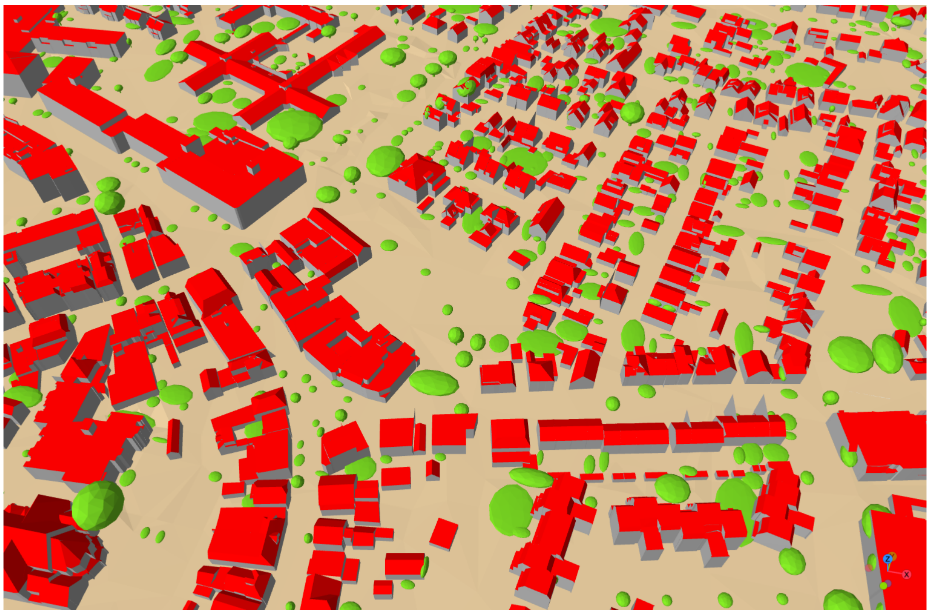
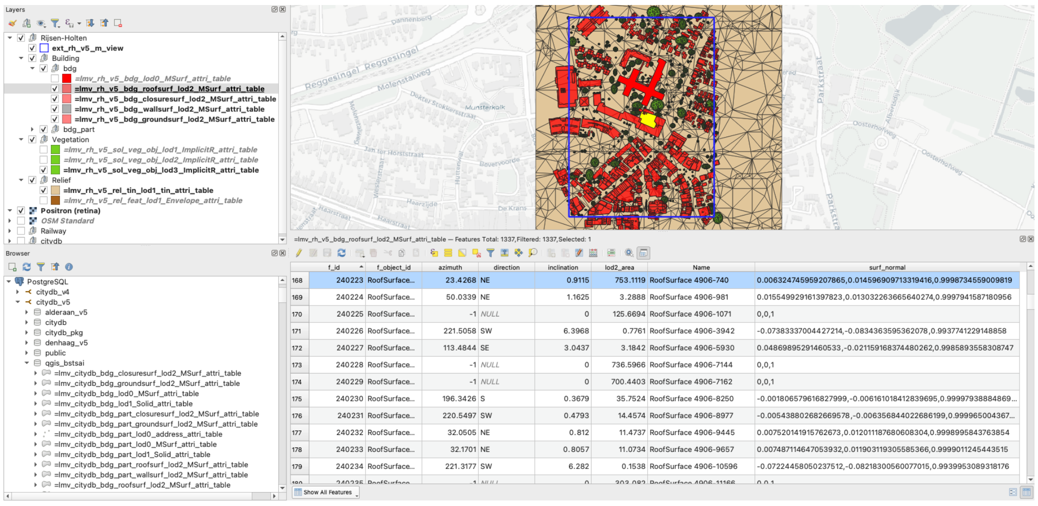


Figure 6.8.: Generated layers of the Rijsen-Holten dataset (full extent) checked in QGIS (3D)



Layers

- Rijsen-Holten
  - ext_rh_v5_m_view
  - Building
    - bdg
      - =lmv_rh_v5_bdg_lod0_MSurf_attri_table
      - =lmv_rh_v5_bdg_roofsurf_lod2_MSurf_attri_table
      - =lmv_rh_v5_bdg_closuresurf_lod2_MSurf_attri_table
      - =lmv_rh_v5_bdg_wallsurf_lod2_MSurf_attri_table
      - =lmv_rh_v5_bdg_groundsurf_lod2_MSurf_attri_table
    - bdg_part
  - Vegetation
    - =lmv_rh_v5_sol_veg_obj_lod1_ImplicitR_attri_table
    - =lmv_rh_v5_sol_veg_obj_lod2_ImplicitR_attri_table
    - =lmv_rh_v5_sol_veg_obj_lod3_ImplicitR_attri_table
  - Relief
    - =lmv_rh_v5_rel_tin_lod1_tin_attri_table
    - =lmv_rh_v5_rel_feat_lod1_Envelope_attri_table
- Positron (retina)
- OSM Standard
- Railway
- citydb

Browser

- PostgreSQL
  - citydb_v4
  - citydb_v5
    - alderaan_v5
    - citydb
    - citydb_pkg
    - denhaag_v5
    - public
    - qgis_bstsai
      - =lmv_citydb_bdg_closuresurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_groundsurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_lod0_MSurf_attri_table
      - =lmv_citydb_bdg_lod1_Solid_attri_table
      - =lmv_citydb_bdg_part_closuresurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_part_groundsurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_part_lod0_address_attri_table
      - =lmv_citydb_bdg_part_lod0_MSurf_attri_table
      - =lmv_citydb_bdg_part_lod1_Solid_attri_table
      - =lmv_citydb_bdg_part_roofsurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_part_wallsurf_lod2_MSurf_attri_table
      - =lmv_citydb_bdg_roofsurf_lod2_MSurf_attri_table

=lmv_rh_v5_bdg_roofsurf_lod2_MSurf_attri_table — Features Total: 1337,Filtered: 1337,Selected: 1

	f_id	f_object_id	azimuth	direction	inclination	lod2_area	Name	surf_normal
168	240223	RoofSurface...	23.4268	NE	0.9115	753.1119	RoofSurface 4906-740	0.006324745959207865,0.014596909713319416,0.9998734559009819
169	240224	RoofSurface...	50.0339	NE	1.1625	3.2888	RoofSurface 4906-981	0.015549929161397823,0.013032263665640274,0.9997941587180956
170	240225	RoofSurface...	-1	NULL	0	125.6694	RoofSurface 4906-1071	0,0,1
171	240226	RoofSurface...	221.5058	SW	6.3968	0.7761	RoofSurface 4906-3942	-0.07383337004427214,-0.0834363595362078,0.9937741229148858
172	240227	RoofSurface...	113.4844	SE	3.0437	3.1842	RoofSurface 4906-5930	0.04869895291460533,-0.021159168374480262,0.9985893558308747
173	240228	RoofSurface...	-1	NULL	0	736.5966	RoofSurface 4906-7144	0,0,1
174	240229	RoofSurface...	-1	NULL	0	700.4403	RoofSurface 4906-7162	0,0,1
175	240230	RoofSurface...	196.3426	S	0.3679	35.7524	RoofSurface 4906-8250	-0.001806579616827999,-0.006161018412839695,0.99997938884869...
176	240231	RoofSurface...	220.5497	SW	0.4793	14.4574	RoofSurface 4906-8977	-0.005438802682669578,-0.006356844022686199,0.999965004367...
177	240232	RoofSurface...	32.0505	NE	0.812	11.4737	RoofSurface 4906-9445	0.007520141915762673,0.012011187680608304,0.9998995843763854
178	240233	RoofSurface...	32.1701	NE	0.8057	11.0734	RoofSurface 4906-9657	0.007487114647053932,0.011903119305585366,0.9999011245443515
179	240234	RoofSurface...	221.3177	SW	6.282	0.1538	RoofSurface 4906-10596	-0.07224458050237512,-0.08218300560077015,0.9939953089318176

Figure 6.9.: Generated layers of the Rijsen-Holten dataset (small extent) checked in QGIS (2D) (User-specified extent coloured in blue; Building roofs layer in LoD2 Multi-Surface selected)

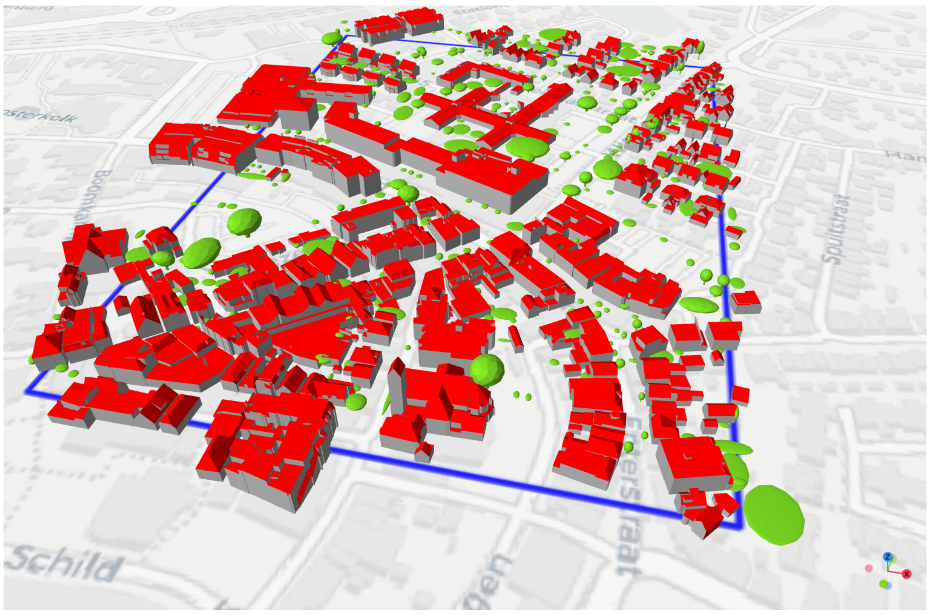
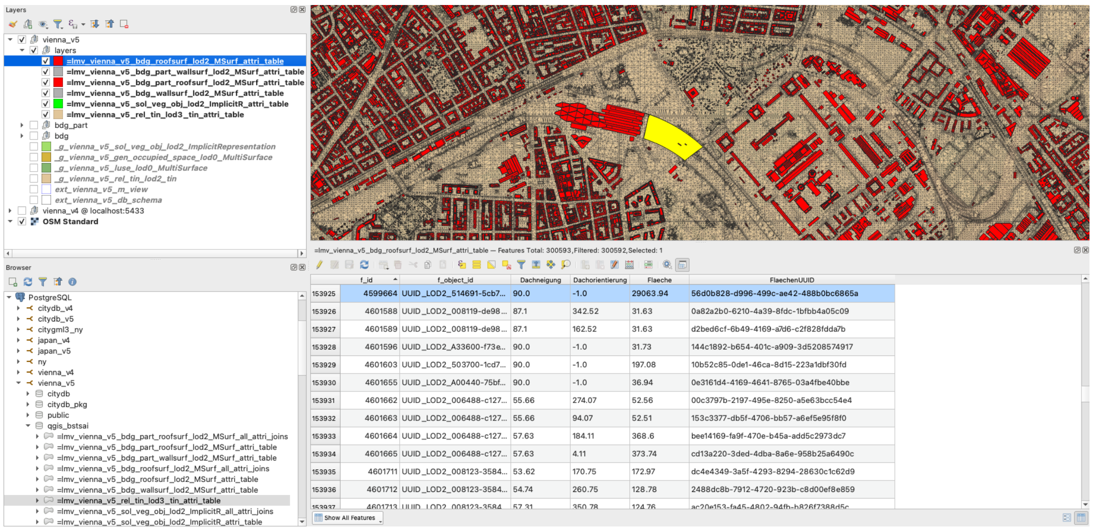


Figure 6.10.: Generated layers of the Rijsen-Holten dataset (small extent) checked in QGIS (3D)

- **Vienna (CityGML2.0)**: The large real-world dataset tests layer generation in an extreme case. It contains classes from commonly seen modules: "Building", "Vegetation", and "Relief". Figure 6.11 presents the full database extent, showing the generation of all existing feature geometries and their attributes. Figure 6.12 displays the zoomed-in result of the layer generation in 3D.



- ✓ vienna_v5
  - ✓ layers
    - ✓ 🟥 =lmv_vienna_v5_bdg_roofsurf_lod2_MSurf_attri_table
    - ✓ ▨ =lmv_vienna_v5_bdg_part_wallsurf_lod2_MSurf_attri_table
    - ✓ 🟥 =lmv_vienna_v5_bdg_part_roofsurf_lod2_MSurf_attri_table
    - ✓ ▨ =lmv_vienna_v5_bdg_wallsurf_lod2_MSurf_attri_table
    - ✓ 🟩 =lmv_vienna_v5_sol_veg_obj_lod2_ImplicitR_attri_table
    - ✓ 🟫 =lmv_vienna_v5_rel_tin_lod3_tin_attri_table
  - bdg_part
  - bdg
  - _g_vienna_v5_sol_veg_obj_lod2_ImplicitRepresentation
  - _g_vienna_v5_gen_occupied_space_lod0_MultiSurface
  - _g_vienna_v5_luse_lod0_MultiSurface
  - _g_vienna_v5_rel_tin_lod2_tin
  - ext_vienna_v5_m_view
  - ext_vienna_v5_db_schema
- vienna_v4 @ localhost:5433
- ✓ OSM Standard

- PostgreSQL
  - citydb_v4
  - citydb_v5
  - citygml3_ny
  - japan_v4
  - japan_v5
  - ny
  - vienna_v4
  - vienna_v5
    - citydb
    - citydb_pkg
    - public
    - qgis_bstsai
      - =lmv_vienna_v5_bdg_part_roofsurf_lod2_MSurf_all_attri_joins
      - =lmv_vienna_v5_bdg_part_roofsurf_lod2_MSurf_attri_table
      - =lmv_vienna_v5_bdg_part_wallsurf_lod2_MSurf_attri_table
      - =lmv_vienna_v5_bdg_roofsurf_lod2_MSurf_all_attri_joins
      - =lmv_vienna_v5_bdg_roofsurf_lod2_MSurf_attri_table
      - =lmv_vienna_v5_bdg_wallsurf_lod2_MSurf_attri_table
      - =lmv_vienna_v5_rel_tin_lod3_tin_attri_table
      - =lmv_vienna_v5_sol_veg_obj_lod2_ImplicitR_all_attri_joins
      - =lmv_vienna_v5_sol_veg_obj_lod2_ImplicitR_attri_table

=lmv_vienna_v5_bdg_roofsurf_lod2_MSurf_attri_table — Features Total: 300593,Filtered: 300592,Selected: 1

	f_id	f_object_id	Dachneigung	Dachorientierung	Flaeche	FlaechenUUID
153925	4599664	UUID_LOD2_514691-5cb7...	90.0	-1.0	29063.94	56d0b828-d996-499c-ae42-488b0bc6865a
153926	4601588	UUID_LOD2_008119-de98...	87.1	342.52	31.63	0a82a2b0-6210-4a39-8fdc-1bfbb4a05c09
153927	4601589	UUID_LOD2_008119-de98...	87.1	162.52	31.63	d2bed6cf-6b49-4169-a7d6-c2f828fdda7b
153928	4601596	UUID_LOD2_A33600-f73e...	90.0	-1.0	31.73	144c1892-b654-401c-a909-3d5208574917
153929	4601603	UUID_LOD2_503700-1cd7...	90.0	-1.0	197.08	10b52c85-0de1-46ca-8d15-223a1dbf30fd
153930	4601655	UUID_LOD2_A00440-75bf...	90.0	-1.0	36.94	0e3161d4-4169-4641-8765-03a4fbe40bbe
153931	4601662	UUID_LOD2_006488-c127...	55.66	274.07	52.56	00c3797b-2197-495e-8250-a5e63bcc54e4
153932	4601663	UUID_LOD2_006488-c127...	55.66	94.07	52.51	153c3377-db5f-4706-bb57-a6ef5e95f8f0
153933	4601664	UUID_LOD2_006488-c127...	57.63	184.11	368.6	bee14169-fa9f-470e-b45a-add5c2973dc7
153934	4601665	UUID_LOD2_006488-c127...	57.63	4.11	373.74	cd13a220-3ded-4dba-8a6e-958b25a6490c
153935	4601711	UUID_LOD2_008123-3584...	53.62	170.75	172.97	dc4e4349-3a5f-4293-8294-28630c1c62d9
153936	4601712	UUID_LOD2_008123-3584...	54.74	260.75	128.78	2488dc8b-7912-4720-923b-c8d00ef8e859
153937	4601713	UUID_LOD2_008123-3584...	57.31	350.78	124.76	ac20e153-fa45-4802-94fb-b826f7388d5c

Figure 6.11.: Generated layers of the Vienna dataset (full extent) checked in QGIS (2D) (Building roofs layer in LoD2 Multi-Surface selected)

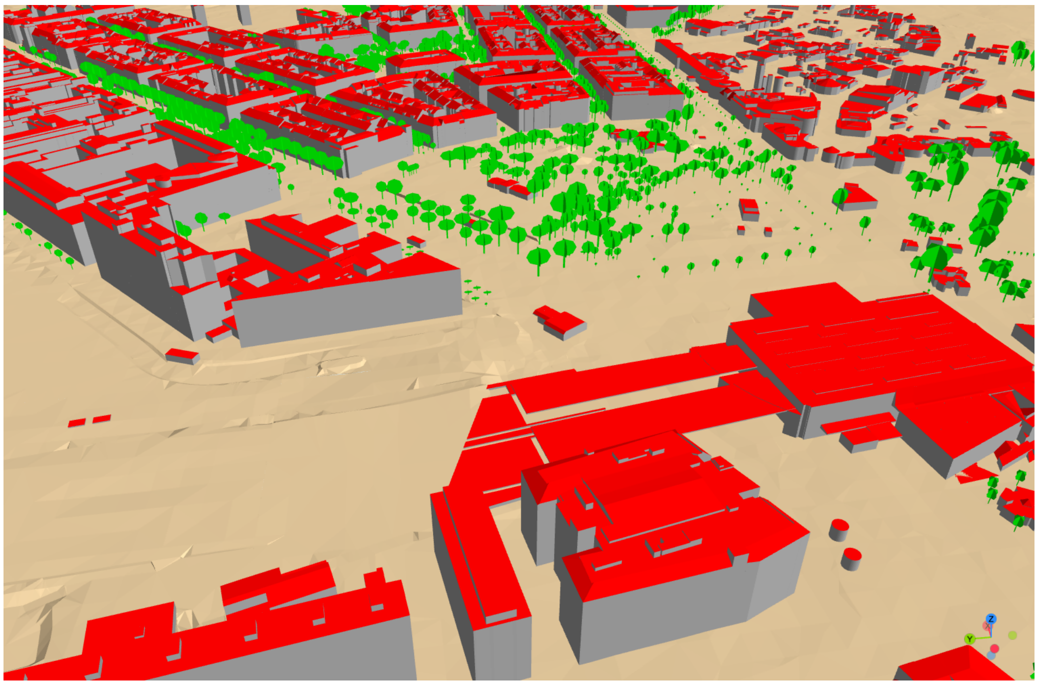
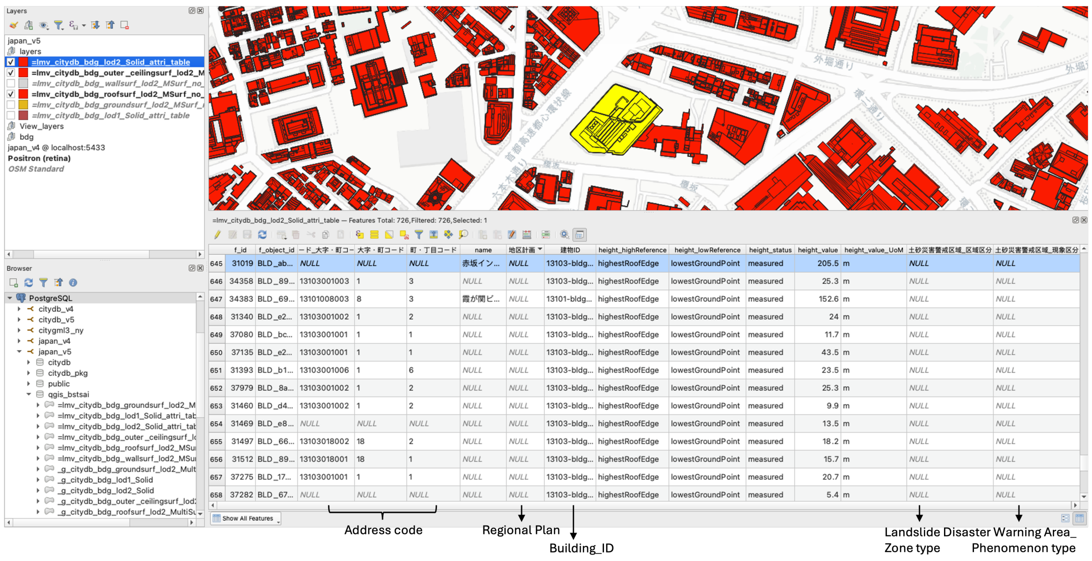


Figure 6.12.: Generated layers of the Vienna dataset (full extent) checked in QGIS (3D)

- **Tokyo (CityGML2.0)**: The real-world dataset is tested using column header names in languages other than English for the generated GIS layers, which contain classes from the "Building" module. Figure 6.13 demonstrates in 2D that the value columns can be renamed using Japanese Kanji characters to represent the flattened feature attributes. Figure 6.14 shows the generated buildings layer in a 3D visualisation.



japan_v5
layers
- [✓] 🟥 =lmv_citydb_bdg_lod2_Solid_attri_table
- [✓] 🟥 =lmv_citydb_bdg_outer _ceilingsurf_lod2_M
- [ ] ▨ =lmv_citydb_bdg_wallsurf_lod2_MSurf_no_
- [✓] 🟥 =lmv_citydb_bdg_roofsurf_lod2_MSurf_no_
- [ ] 🟨 =lmv_citydb_bdg_groundsurf_lod2_MSurf_
- [ ] 🟥 =lmv_citydb_bdg_lod1_Solid_attri_table

View_layers
bdg
japan_v4 @ localhost:5433
Positron (retina)
OSM Standard

```
PostgreSQL
  citydb_v4
  citydb_v5
  citygml3_ny
  japan_v4
  japan_v5
    citydb
    citydb_pkg
    public
    qgis_bstsai
      =lmv_citydb_bdg_groundsurf_lod2_M
      =lmv_citydb_bdg_lod1_Solid_attri_tab
      =lmv_citydb_bdg_lod2_Solid_attri_tab
      =lmv_citydb_bdg_outer _ceilingsurf_l
      =lmv_citydb_bdg_roofsurf_lod2_MSu
      =lmv_citydb_bdg_wallsurf_lod2_MSu
      _g_citydb_bdg_groundsurf_lod2_Mult
      _g_citydb_bdg_lod1_Solid
      _g_citydb_bdg_lod2_Solid
      _g_citydb_bdg_outer _ceilingsurf_lod
      _g_citydb_bdg_roofsurf_lod2_MultiSu
```

=lmv_citydb_bdg_lod2_Solid_attri_table — Features Total: 726,Filtered: 726,Selected: 1

	f_id	f_object_id	ード_大字・町コー	大字・町コード	町・丁目コード	name	地区計画	建物ID	height_highReference	height_lowReference	height_status	height_value	height_value_UoM	土砂災害警戒区域_区域区分	土砂災害警戒区域_現象区分
645	31019	BLD_ab...	NULL	NULL	NULL	赤坂イン...	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	205.5	m	NULL	NULL
646	34358	BLD_89...	13103001003	1	3	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	25.3	m	NULL	NULL
647	34383	BLD_69...	13101008003	8	3	霞が関ビ...	NULL	13101-bldg...	highestRoofEdge	lowestGroundPoint	measured	152.6	m	NULL	NULL
648	31340	BLD_e2...	13103001002	1	2	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	24	m	NULL	NULL
649	37080	BLD_bc...	13103001001	1	1	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	11.7	m	NULL	NULL
650	37135	BLD_e2...	13103001001	1	1	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	43.5	m	NULL	NULL
651	31393	BLD_b1...	13103001006	1	6	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	23.5	m	NULL	NULL
652	37979	BLD_8a...	13103001002	1	2	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	25.3	m	NULL	NULL
653	31460	BLD_d4...	13103001002	1	2	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	9.9	m	NULL	NULL
654	31469	BLD_e8...	NULL	NULL	NULL	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	13.5	m	NULL	NULL
655	31497	BLD_66...	13103018002	18	2	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	18.2	m	NULL	NULL
656	31512	BLD_89...	13103018001	18	1	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	15.7	m	NULL	NULL
657	37275	BLD_17...	13103001001	1	1	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	20.7	m	NULL	NULL
658	37282	BLD_67...	NULL	NULL	NULL	NULL	NULL	13103-bldg...	highestRoofEdge	lowestGroundPoint	measured	5.4	m	NULL	NULL

Figure 6.13.: Generated layers of the Tokyo dataset (one file full extent) checked in QGIS (2D) (Buildings layer in LoD2 Solid selected)

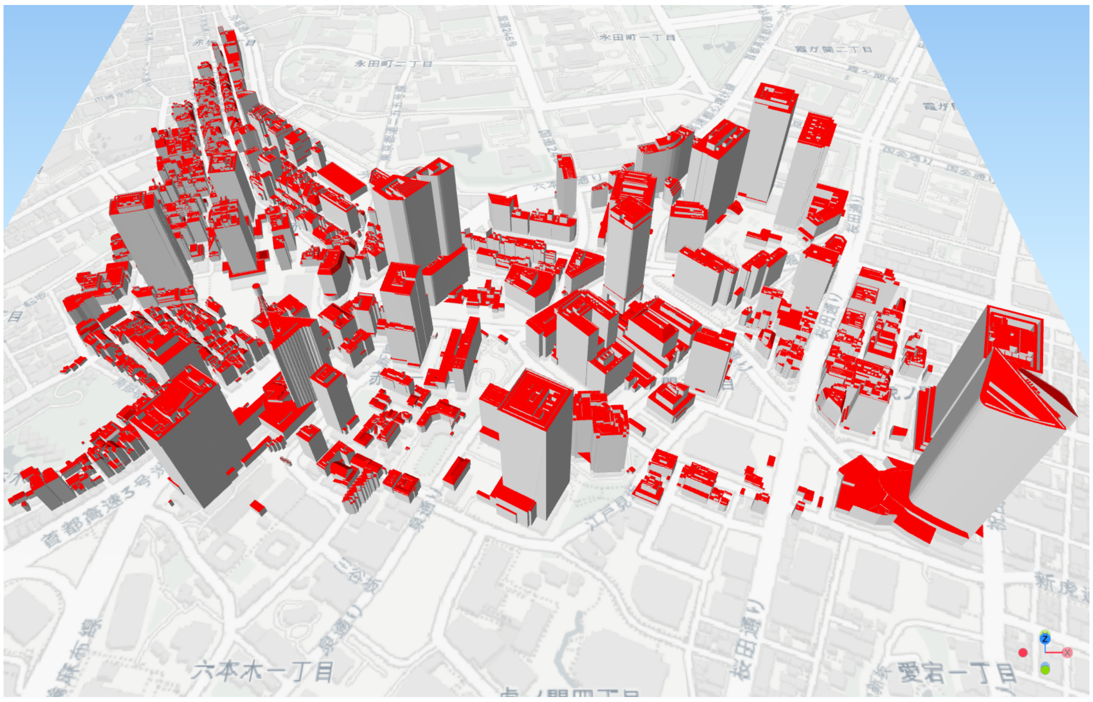
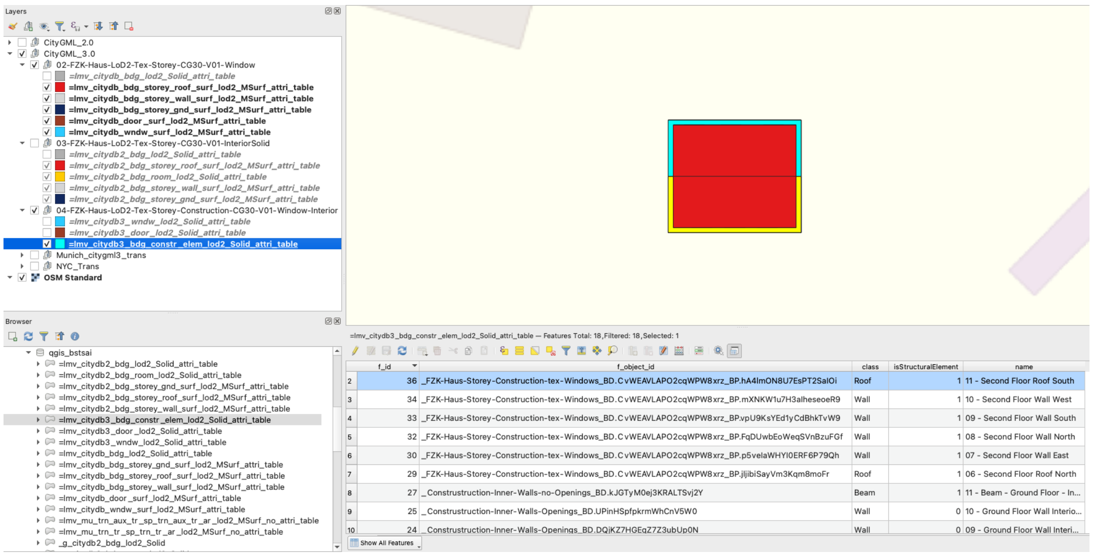


Figure 6.14.: Generated layers of the Tokyo dataset (one file full extent) checked in QGIS (3D)

- **FZK-Haus (CityGML3.0)**: This artificial dataset is used to test GIS layer generation from CityGML v.3.0 data, which contains classes from the "Building" and "Construction" module (Figure 6.15). The test dataset files downloaded from [42] include building construction elements (roofs, walls, Beam, etc.), storeys and interior rooms. These are the distinct features introduced in CityGML v.3.0 standards to enhance interoperability with IFC objects for supporting building information management and can now be represented in different LoDs and geometries, e.g. building storeys in LoD2 Multi-Surface (Figure 6.16a) and building rooms in LoD2 Solid (Figure 6.16b).


Layers

- ☐ CityGML_2.0
- ✓ CityGML_3.0
  - ✓ 02-FZK-Haus-LoD2-Tex-Storey-CG30-V01-Window
    - ☐ =lmv_citydb_bdg_lod2_Solid_attri_table
    - ✓ 🟥 =lmv_citydb_bdg_storey_roof_surf_lod2_MSurf_attri_table
    - ✓ =lmv_citydb_bdg_storey_wall_surf_lod2_MSurf_attri_table
    - ✓ =lmv_citydb_bdg_storey_gnd_surf_lod2_MSurf_attri_table
    - ✓ =lmv_citydb_door_surf_lod2_MSurf_attri_table
    - ✓ =lmv_citydb_wndw_surf_lod2_MSurf_attri_table
  - ☐ 03-FZK-Haus-LoD2-Tex-Storey-CG30-V01-InteriorSolid
    - ☐ =lmv_citydb2_bdg_lod2_Solid_attri_table
    - ✓ 🟥 =lmv_citydb2_bdg_storey_roof_surf_lod2_MSurf_attri_table
    - ✓ 🟨 =lmv_citydb2_bdg_room_lod2_Solid_attri_table
    - ✓ =lmv_citydb2_bdg_storey_wall_surf_lod2_MSurf_attri_table
    - ✓ =lmv_citydb2_bdg_storey_gnd_surf_lod2_MSurf_attri_table
  - ✓ 04-FZK-Haus-LoD2-Tex-Storey-Construction-CG30-V01-Window-Interior
    - ☐ =lmv_citydb3_wndw_lod2_Solid_attri_table
    - ☐ =lmv_citydb3_door_lod2_Solid_attri_table
    - ✓ =lmv_citydb3_bdg_constr_elem_lod2_Solid_attri_table
  - ☐ Munich_citygml3_trans
  - ☐ NYC_Trans
- ✓ OSM Standard

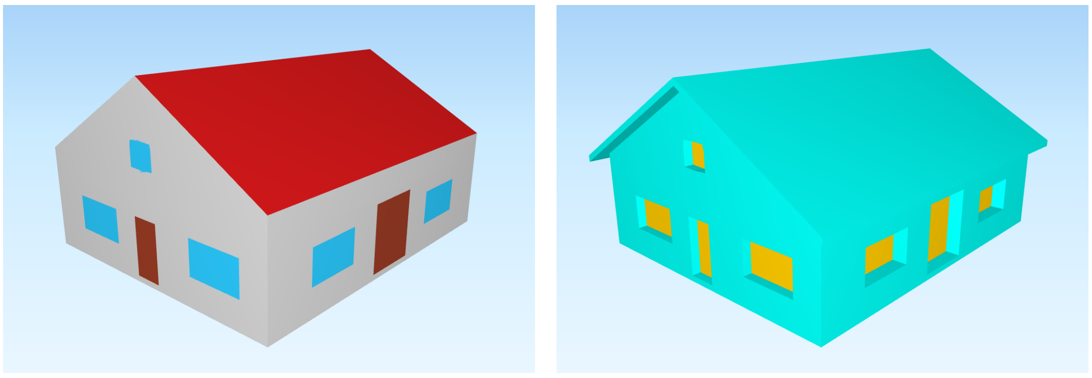
Browser

- qgis_bstsai
  - =lmv_citydb2_bdg_lod2_Solid_attri_table
  - =lmv_citydb2_bdg_room_lod2_Solid_attri_table
  - =lmv_citydb2_bdg_storey_gnd_surf_lod2_MSurf_attri_table
  - =lmv_citydb2_bdg_storey_roof_surf_lod2_MSurf_attri_table
  - =lmv_citydb2_bdg_storey_wall_surf_lod2_MSurf_attri_table
  - =lmv_citydb3_bdg_constr_elem_lod2_Solid_attri_table
  - =lmv_citydb3_door_lod2_Solid_attri_table
  - =lmv_citydb3_wndw_lod2_Solid_attri_table
  - =lmv_citydb_bdg_lod2_Solid_attri_table
  - =lmv_citydb_bdg_storey_gnd_surf_lod2_MSurf_attri_table
  - =lmv_citydb_bdg_storey_roof_surf_lod2_MSurf_attri_table
  - =lmv_citydb_bdg_storey_wall_surf_lod2_MSurf_attri_table
  - =lmv_citydb_door_surf_lod2_MSurf_attri_table
  - =lmv_citydb_wndw_surf_lod2_MSurf_attri_table
  - =lmv_mu_trn_aux_tr_sp_trn_aux_tr_ar_lod2_MSurf_no_attri_table
  - =lmv_mu_trn_tr_sp_trn_tr_ar_lod2_MSurf_no_attri_table
  - _g_citydb2_bdg_lod2_Solid

=lmv_citydb3_bdg_constr_elem_lod2_Solid_attri_table — Features Total: 18, Filtered: 18, Selected: 1

	f_id	f_object_id	class	isStructuralElement	name
2	36	_FZK-Haus-Storey-Construction-tex-Windows_BD.CvWEAVLAPO2cqWPW8xrz_BP.hA4lmON8U7EsPT2SalOi	Roof	1	11 - Second Floor Roof South
3	34	_FZK-Haus-Storey-Construction-tex-Windows_BD.CvWEAVLAPO2cqWPW8xrz_BP.mXNKW1u7H3alheseoeR9	Wall	1	10 - Second Floor Wall West
4	33	_FZK-Haus-Storey-Construction-tex-Windows_BD.CvWEAVLAPO2cqWPW8xrz_BP.vpU9KsYEd1yCdBhkTvW9	Wall	1	09 - Second Floor Wall South
5	32	_FZK-Haus-Storey-Construction-tex-Windows_BD.CvWEAVLAPO2cqWPW8xrz_BP.FqDUwbEoWeqSVnBzuFGf	Wall	1	08 - Second Floor Wall North
6	30	_FZK-Haus-Storey-Construction-tex-Windows_BD.CvWEAVLAPO2cqWPW8xrz_BP.p5velaWHYI0ERF6P79Qh	Wall	1	07 - Second Floor Wall East
7	29	_FZK-Haus-Storey-Construction-tex-Windows_BD.CvWEAVLAPO2cqWPW8xrz_BP.jljibiSayVm3Kqm8moFr	Roof	1	06 - Second Floor Roof North
8	27	_Construstruction-Inner-Walls-no-Openings_BD.kJGTyM0ej3KRALTSvj2Y	Beam	1	11 - Beam - Ground Floor - In...
9	25	_Construstruction-Inner-Walls-Openings_BD.UPinHSpfpkrmWhCnV5W0	Wall	0	10 - Ground Floor Wall Interio...
10	24	_Construstruction-Inner-Walls-Openings_BD.DQjKZ7HGEqZ7Z3ubUp0N	Wall	0	09 - Ground Floor Wall Interi...

Figure 6.15.: Generated layers of the FZK-Haus dataset (full extent) checked in QGIS (2D) (Building construction elements layer in LoD2 Solid selected)

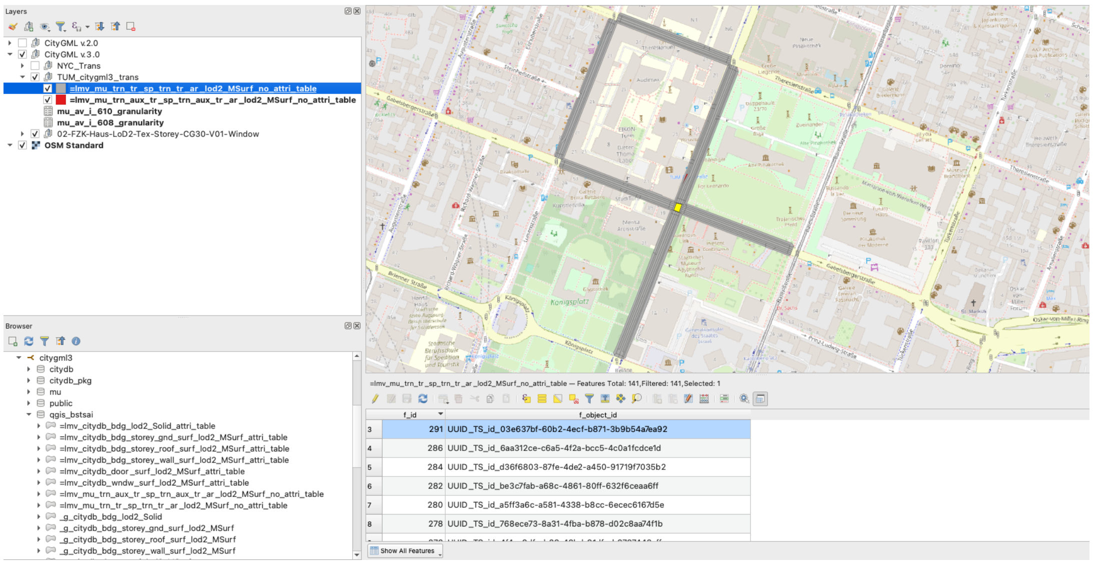

(a) Building storey roofs (red), walls (grey) with windows (blue) and doors (brown)

(b) Building construction elements (blue) and interior rooms (yellow)

Figure 6.16.: Generated layers of the FZK-Haus dataset (full extent) checked in QGIS (3D)

- **Munich and New York City Transportation (CityGML3.0)**: These two real-world datasets are used to test GIS layer generation from classes in the modified "Transportation" module following the CityGML v.3.0 standards. Figure 6.17 and Figure 6.18 show the 2D and 3D visualisations of the GIS layers generated from the Munich transportation dataset, while Figure 6.19 and Figure 6.20 present results from the New York City transportation dataset. These datasets include features such as (auxiliary) traffic spaces and areas, roads, sections, intersections, and squares.

  The concept of (auxiliary) traffic space is introduced in CityGML v.3.0 to represent 3D spaces where traffic occurs, in addition to surface-based objects. However, only the (auxiliary) traffic areas, the child features of the (auxiliary) traffic space, have direct LoD and geometry representations in these datasets. As a result, the GIS layers are generated without attributes since the (auxiliary) traffic spaces have no direct geometry views available to associate their attributes for visualisation. Additionally, features such as roads, sections, intersections, and squares do not have any direct LoDs and geometry representations. They only have feature relation properties to the (auxiliary) traffic spaces and areas as nested features, making them currently unavailable via GIS layers. Further investigation is required to address these limitations in the future development of the plug-in, and certain possible solutions are elaborated in Section 8.2.2 and Section 8.2.3.



Layers

- CityGML v.2.0
- ✓ CityGML v.3.0
  - NYC_Trans
  - ✓ TUM_citygml3_trans
    - ✓ =lmv_mu_trn_tr_sp_trn_tr_ar_lod2_MSurf_no_attri_table
    - ✓ =lmv_mu_trn_aux_tr_sp_trn_aux_tr_ar_lod2_MSurf_no_attri_table
    - mu_av_i_610_granularity
    - mu_av_i_608_granularity
  - ✓ 02-FZK-Haus-LoD2-Tex-Storey-CG30-V01-Window
- ✓ OSM Standard

Browser

- citygml3
  - citydb
  - citydb_pkg
  - mu
  - public
  - qgis_bstsai
    - =lmv_citydb_bdg_lod2_Solid_attri_table
    - =lmv_citydb_bdg_storey_gnd_surf_lod2_MSurf_attri_table
    - =lmv_citydb_bdg_storey_roof_surf_lod2_MSurf_attri_table
    - =lmv_citydb_bdg_storey_wall_surf_lod2_MSurf_attri_table
    - =lmv_citydb_door_surf_lod2_MSurf_attri_table
    - =lmv_citydb_wndw_surf_lod2_MSurf_attri_table
    - =lmv_mu_trn_aux_tr_sp_trn_aux_tr_ar_lod2_MSurf_no_attri_table
    - =lmv_mu_trn_tr_sp_trn_tr_ar_lod2_MSurf_no_attri_table
    - _g_citydb_bdg_lod2_Solid
    - _g_citydb_bdg_storey_gnd_surf_lod2_MSurf
    - _g_citydb_bdg_storey_roof_surf_lod2_MSurf
    - _g_citydb_bdg_storey_wall_surf_lod2_MSurf

=lmv_mu_trn_tr_sp_trn_tr_ar_lod2_MSurf_no_attri_table — Features Total: 141,Filtered: 141,Selected: 1

	f_id	f_object_id
3	291	UUID_TS_id_03e637bf-60b2-4ecf-b871-3b9b54a7ea92
4	286	UUID_TS_id_6aa312ce-c6a5-4f2a-bcc5-4c0a1fcdce1d
5	284	UUID_TS_id_d36f6803-87fe-4de2-a450-91719f7035b2
6	282	UUID_TS_id_be3c7fab-a68c-4861-80ff-632f6ceaa6ff
7	280	UUID_TS_id_a5ff3a6c-a581-4338-b8cc-6ecec6167d5e
8	278	UUID_TS_id_768ece73-8a31-4fba-b878-d02c8aa74f1b

Figure 6.17.: Generated layers of the Munich transportation dataset (full extent) checked in QGIS (2D)
(Traffic space traffic areas layer in LoD2 Multi-Surface selected)

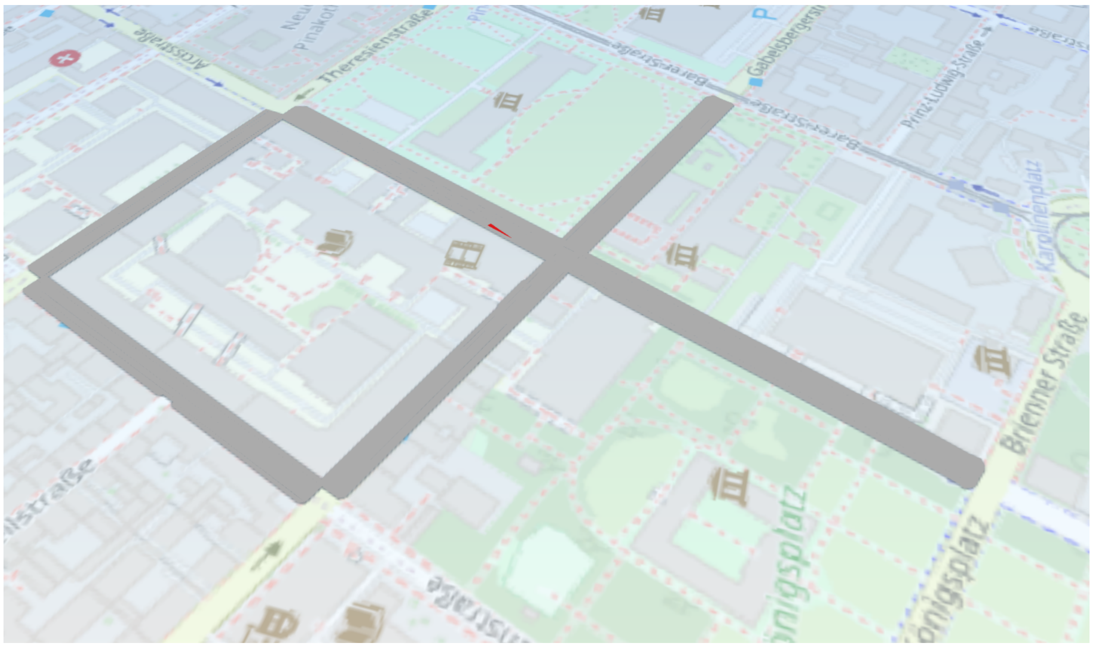
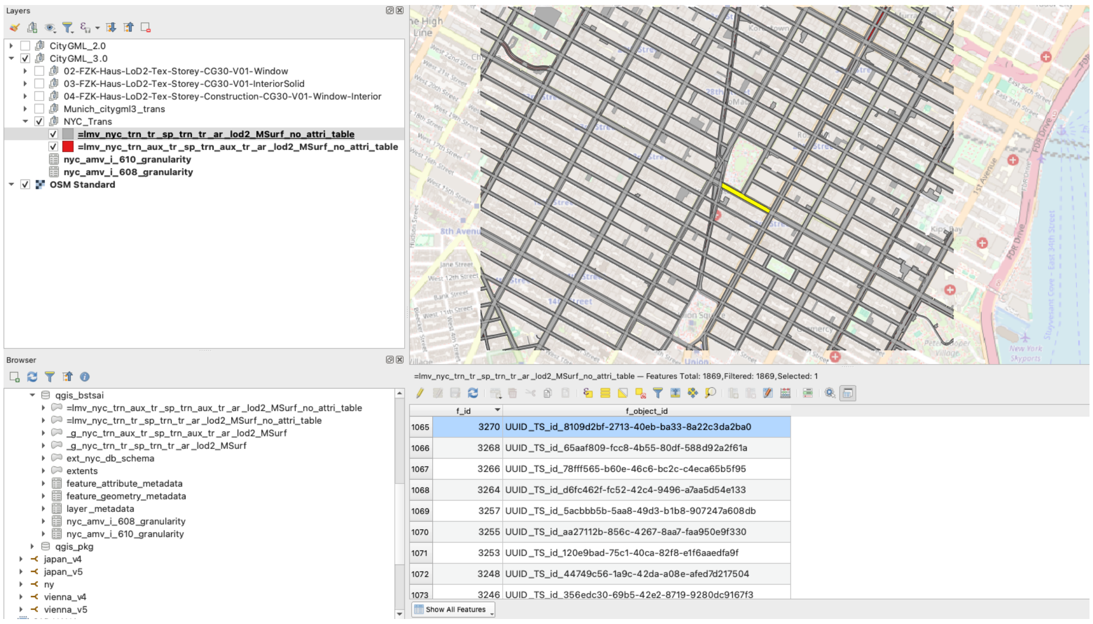


Figure 6.18.: Generated layers of the Munich transportation (full extent) checked in QGIS (3D) (Traffic areas in grey and auxiliary traffic areas in red)



```
Layers
CityGML_2.0
CityGML_3.0
    02-FZK-Haus-LoD2-Tex-Storey-CG30-V01-Window
    03-FZK-Haus-LoD2-Tex-Storey-CG30-V01-InteriorSolid
    04-FZK-Haus-LoD2-Tex-Storey-Construction-CG30-V01-Window-Interior
    Munich_citygml3_trans
    NYC_Trans
        =lmv_nyc_trn_tr_sp_trn_tr_ar_lod2_MSurf_no_attri_table
        =lmv_nyc_trn_aux_tr_sp_trn_aux_tr_ar_lod2_MSurf_no_attri_table
        nyc_amv_i_610_granularity
        nyc_amv_i_608_granularity
OSM Standard

Browser
    qgis_bstsai
        =lmv_nyc_trn_aux_tr_sp_trn_aux_tr_ar_lod2_MSurf_no_attri_table
        =lmv_nyc_trn_tr_sp_trn_tr_ar_lod2_MSurf_no_attri_table
        _g_nyc_trn_aux_tr_sp_trn_aux_tr_ar_lod2_MSurf
        _g_nyc_trn_tr_sp_trn_tr_ar_lod2_MSurf
        ext_nyc_db_schema
        extents
        feature_attribute_metadata
        feature_geometry_metadata
        layer_metadata
        nyc_amv_i_608_granularity
        nyc_amv_i_610_granularity
    qgis_pkg
japan_v4
japan_v5
ny
vienna_v4
vienna_v5
```

=lmv_nyc_trn_tr_sp_trn_tr_ar_lod2_MSurf_no_attri_table — Features Total: 1869,Filtered: 1869,Selected: 1

	f_id	f_object_id
1065	3270	UUID_TS_id_8109d2bf-2713-40eb-ba33-8a22c3da2ba0
1066	3268	UUID_TS_id_65aaf809-fcc8-4b55-80df-588d92a2f61a
1067	3266	UUID_TS_id_78fff565-b60e-46c6-bc2c-c4eca65b5f95
1068	3264	UUID_TS_id_d6fc462f-fc52-42c4-9496-a7aa5d54e133
1069	3257	UUID_TS_id_5acbbb5b-5aa8-49d3-b1b8-907247a608db
1070	3255	UUID_TS_id_aa27112b-856c-4267-8aa7-faa950e9f330
1071	3253	UUID_TS_id_120e9bad-75c1-40ca-82f8-e1f6aaedfa9f
1072	3248	UUID_TS_id_44749c56-1a9c-42da-a08e-afed7d217504
1073	3246	UUID_TS_id_356edc30-69b5-42e2-8719-9280dc9167f3

Figure 6.19.: Generated layers of the New York City transportation dataset (full extent) checked in QGIS (2D)
(Traffic space traffic areas layer in LoD2 Multi-Surface selected)

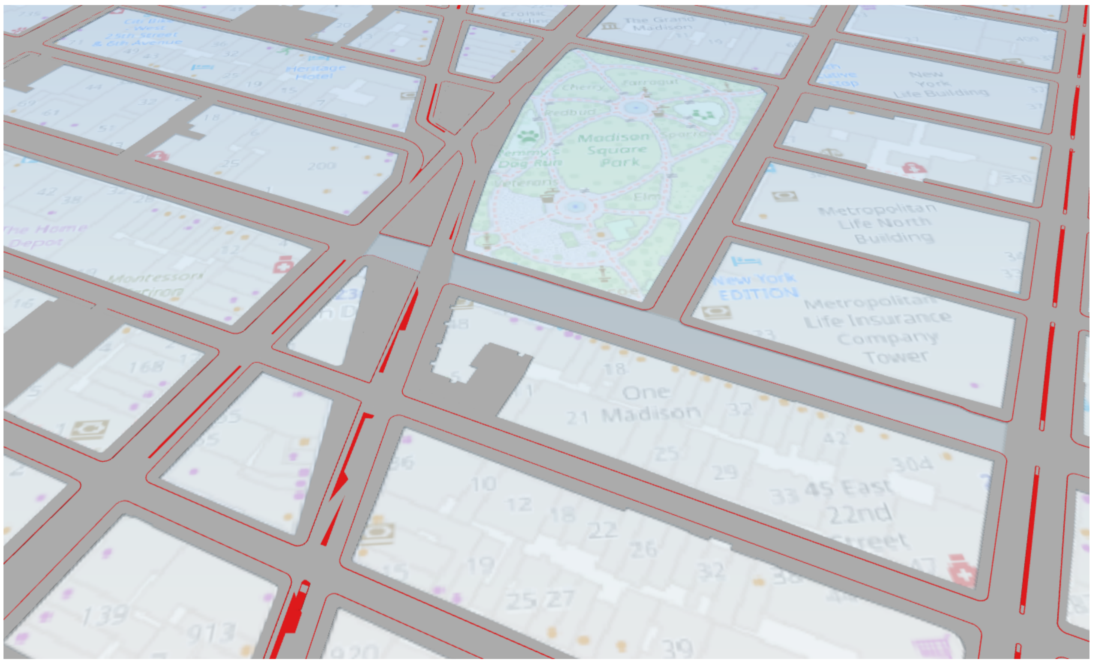


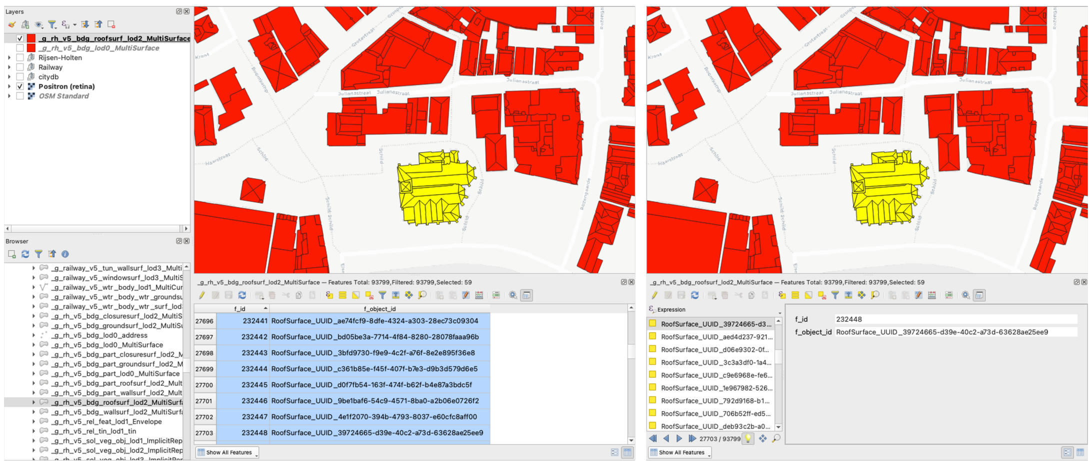
Figure 6.20.: Generated layers of the New York City Transportation (full extent) checked in QGIS (3D)
(Traffic areas in grey and auxiliary traffic areas in red)

7. Evaluation of Possible Use Cases for CityGML Data in QGIS

With the methodology and implementation for collecting and processing feature geometries and attributes detailed in Chapter 4, Chapter 5, and Chapter 6, users can now create GIS layers from data stored in 3DCityDB v.5.0. This chapter evaluates the extent to which this research fulfils the possible use cases identified in Chapter 3.

7.1. Case 1: Users interact Only with feature geometries

Users can collect feature geometries and store them in materialized views to visualise the results in QGIS by the following Section 4.2 and Section 5.3. Each feature geometry with its unique `feature_id(f_id)` and `f_object_id` can then be inspected in 2D using either the "attribute table" or "attribute form" in QGIS (Figure 7.1, the selected feature is coloured in yellow). Users can visualise feature geometries in 3D using either the QGIS 3D Map or the plug-in called "Qgis2threejs" (Figure 7.2).


Layers
_g_rh_v5_bdg_roofsurf_lod2_MultiSurface
_g_rh_v5_bdg_lod0_MultiSurface
Rijsen-Holten
Railway
citydb
Positron (retina)
OSM Standard
Browser
_g_railway_v5_tun_wallsurf_lod3_Multi
_g_railway_v5_windowsurf_lod3_MultiS
_g_railway_v5_wtr_body_lod1_MultiCur
_g_railway_v5_wtr_body_wtr_groundsu
_g_railway_v5_wtr_body_wtr_surf_lod
_g_rh_v5_bdg_closuresurf_lod2_MultiS
_g_rh_v5_bdg_groundsurf_lod2_MultiSu
_g_rh_v5_bdg_lod0_address
_g_rh_v5_bdg_lod0_MultiSurface
_g_rh_v5_bdg_part_closuresurf_lod2_M
_g_rh_v5_bdg_part_groundsurf_lod2_M
_g_rh_v5_bdg_part_lod0_MultiSurface
_g_rh_v5_bdg_part_roofsurf_lod2_Mult
_g_rh_v5_bdg_part_wallsurf_lod2_Mult
_g_rh_v5_bdg_roofsurf_lod2_MultiSurfa
_g_rh_v5_bdg_wallsurf_lod2_MultiSurfa
_g_rh_v5_rel_feat_lod1_Envelope
_g_rh_v5_rel_tin_lod1_tin
_g_rh_v5_sol_veg_obj_lod1_ImplicitRep
_g_rh_v5_sol_veg_obj_lod2_ImplicitRep
Show All Features
Expression

_g_rh_v5_bdg_roofsurf_lod2_MultiSurface — Features Total: 93799,Filtered: 93799,Selected: 59

	f_id	f_object_id
27696	232441	RoofSurface_UUID_ae74fcf9-8dfe-4324-a303-28ec73c09304
27697	232442	RoofSurface_UUID_bd05be3a-7714-4f84-8280-28078faaa96b
27698	232443	RoofSurface_UUID_3bfd9730-f9e9-4c2f-a76f-8e2e895f36e8
27699	232444	RoofSurface_UUID_c361b85e-f45f-407f-b7e3-d9b3d579d6e5
27700	232445	RoofSurface_UUID_d0f7fb54-163f-474f-b62f-b4e87a3bdc5f
27701	232446	RoofSurface_UUID_9be1baf6-54c9-4571-8ba0-a2b06e0726f2
27702	232447	RoofSurface_UUID_4e1f2070-394b-4793-8037-e60cfc8aff00
27703	232448	RoofSurface_UUID_39724665-d39e-40c2-a73d-63628ae25ee9

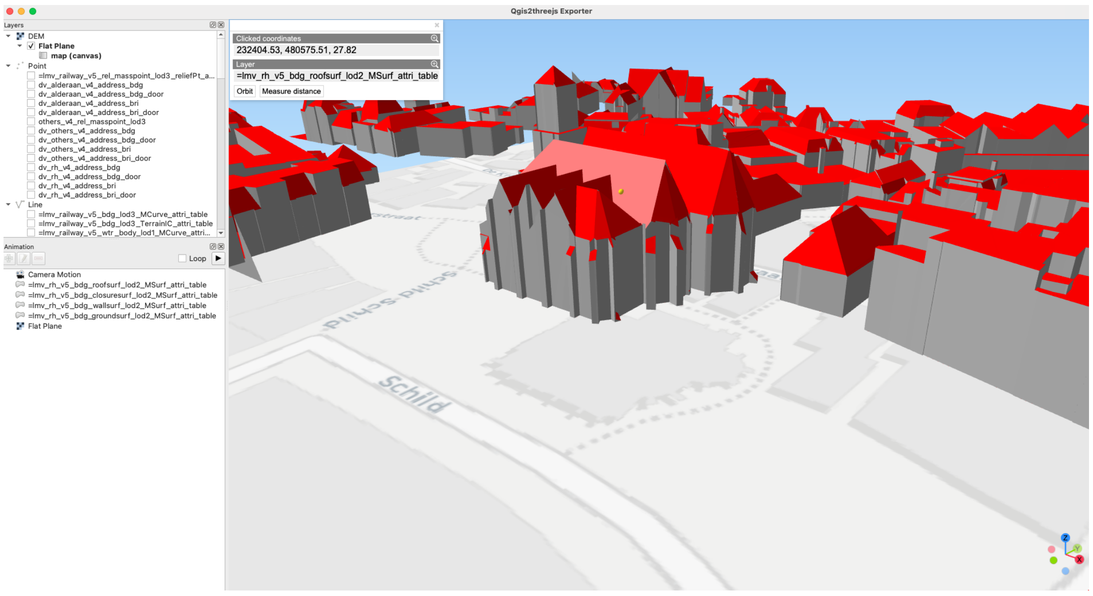
_g_rh_v5_bdg_roofsurf_lod2_MultiSurface — Features Total: 93799,Filtered: 93799,Selected: 59

RoofSurface_UUID_39724665-d3...
RoofSurface_UUID_aed4d237-921...
RoofSurface_UUID_d06e9302-0f...
RoofSurface_UUID_3c3a3df0-1a4...
RoofSurface_UUID_c9e6968e-fe6...
RoofSurface_UUID_1e967982-526...
RoofSurface_UUID_792d9168-b1...
RoofSurface_UUID_706b52ff-ed5...
RoofSurface_UUID_deb93c2b-a0...

27703 / 93799

f_id	232448
f_object_id	RoofSurface_UUID_39724665-d39e-40c2-a73d-63628ae25ee9

Figure 7.1.: Use case 1 checked - feature geometries viewed in 2D (Left) attribute table view. (Right) attribute form view

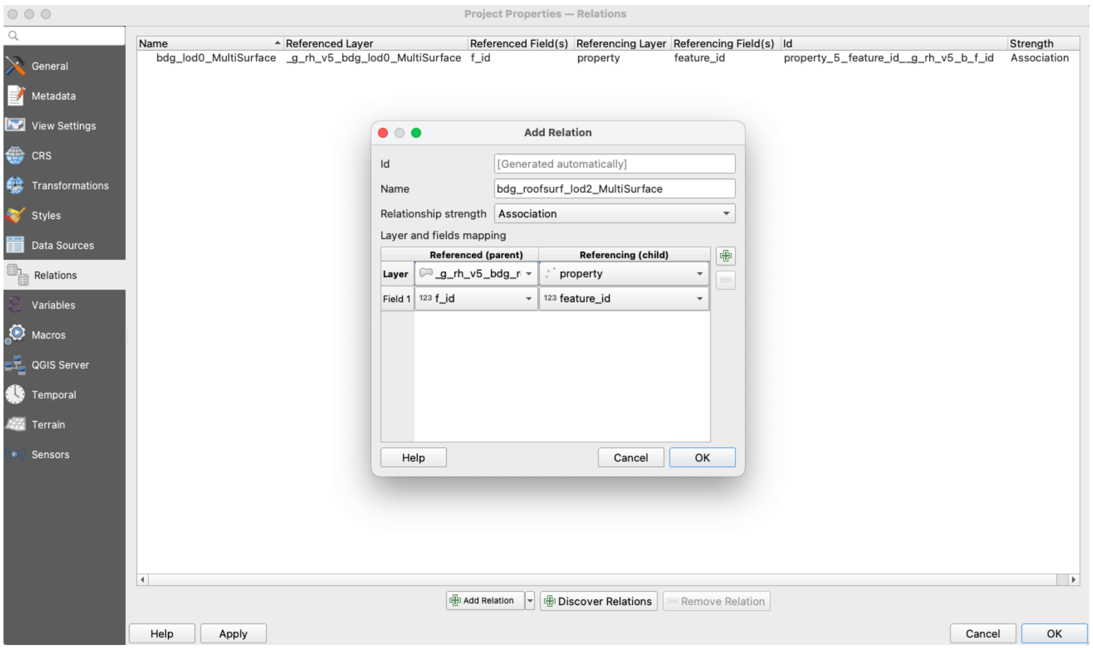

Qgis2threejs Exporter
Layers
DEM
Flat Plane
map (canvas)
Point
=lmv_railway_v5_rel_masspoint_lod3_reliefPt_a...
dv_alderaan_v4_address_bdg
dv_alderaan_v4_address_bdg_door
dv_alderaan_v4_address_bri
dv_alderaan_v4_address_bri_door
others_v4_rel_masspoint_lod3
dv_others_v4_address_bdg
dv_others_v4_address_bdg_door
dv_others_v4_address_bri
dv_others_v4_address_bri_door
dv_rh_v4_address_bdg
dv_rh_v4_address_bdg_door
dv_rh_v4_address_bri
dv_rh_v4_address_bri_door
Line
=lmv_railway_v5_bdg_lod3_MCurve_attri_table
=lmv_railway_v5_bdg_lod3_TerrainIC_attri_table
=lmv_railway_v5_wtr_body_lod1_MCurve_attri...
Animation
Loop
Camera Motion
=lmv_rh_v5_bdg_roofsurf_lod2_MSurf_attri_table
=lmv_rh_v5_bdg_closuresurf_lod2_MSurf_attri_table
=lmv_rh_v5_bdg_wallsurf_lod2_MSurf_attri_table
=lmv_rh_v5_bdg_groundsurf_lod2_MSurf_attri_table
Flat Plane
Orbit
Measure distance
Schild

Clicked coordinates
232404.53, 480575.51, 27.82
Layer
=lmv_rh_v5_bdg_roofsurf_lod2_MSurf_attri_table

Figure 7.2.: Use case 1 checked - feature geometries viewed in 3D (Qgis2threejs plug-in)

7.2. Case 2: Users Interact with Visualised Feature Geometries and Retrieve the Attributes by Clicking on Features

The `cdb_schema` PROPERTY table can be imported into QGIS to be associated with the selected feature geometry materialized view. Figure 7.3 shows the process of adding relations between the `cdb_schema` PROPERTY table and the buildings materialized view in LoD0 multi-surface. By joining the `f_id` of the buildings with the `feature_id` of the property table, users can view the associated attributes in the "attribute form" (Figure 7.4). The feature attributes can only be edited in the EAV model storage due to the structure of the `cdb_schema` PROPERTY table. Users can query the target attribute name and change the values within the corresponding value columns (Figure 7.5). The two sub-use cases, i.e. read-only and read-write, can be achieved by manually adding relations with the feature geometry materialized views generated by the server-side plug-in developed in this research.


Project Properties — Relations

Name	Referenced Layer	Referenced Field(s)	Referencing Layer	Referencing Field(s)	Id	Strength
bdg_lod0_MultiSurface	_g_rh_v5_bdg_lod0_MultiSurface	f_id	property	feature_id	property_5_feature_id__g_rh_v5_b_f_id	Association

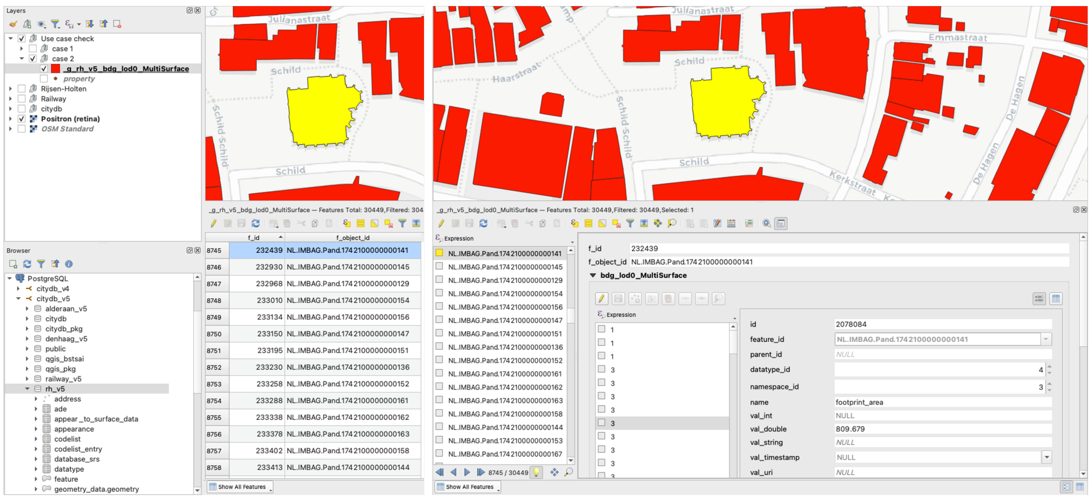
Add Relation

Id	[Generated automatically]
Name	bdg_roofsurf_lod2_MultiSurface
Relationship strength	Association

Layer and fields mapping

	Referenced (parent)	Referencing (child)
Layer	_g_rh_v5_bdg_r	property
Field 1	f_id	feature_id

Figure 7.3.: Use case 2 checked - Adding relations of feature attributes in QGIS


_g_rh_v5_bdg_lod0_MultiSurface — Features Total: 30449,Filtered: 304

	f_id	f_object_id
8745	232439	NL.IMBAG.Pand.1742100000000141
8746	232930	NL.IMBAG.Pand.1742100000000145
8747	232968	NL.IMBAG.Pand.1742100000000129
8748	233010	NL.IMBAG.Pand.1742100000000154
8749	233134	NL.IMBAG.Pand.1742100000000156
8750	233150	NL.IMBAG.Pand.1742100000000147
8751	233195	NL.IMBAG.Pand.1742100000000151
8752	233230	NL.IMBAG.Pand.1742100000000136
8753	233258	NL.IMBAG.Pand.1742100000000152
8754	233288	NL.IMBAG.Pand.1742100000000161
8755	233338	NL.IMBAG.Pand.1742100000000162
8756	233378	NL.IMBAG.Pand.1742100000000163
8757	233402	NL.IMBAG.Pand.1742100000000158
8758	233413	NL.IMBAG.Pand.1742100000000144

_g_rh_v5_bdg_lod0_MultiSurface — Features Total: 30449,Filtered: 30449,Selected: 1

NL.IMBAG.Pand.1742100000000141
NL.IMBAG.Pand.1742100000000145
NL.IMBAG.Pand.1742100000000129
NL.IMBAG.Pand.1742100000000154
NL.IMBAG.Pand.1742100000000156
NL.IMBAG.Pand.1742100000000147
NL.IMBAG.Pand.1742100000000151
NL.IMBAG.Pand.1742100000000136
NL.IMBAG.Pand.1742100000000152
NL.IMBAG.Pand.1742100000000161
NL.IMBAG.Pand.1742100000000162
NL.IMBAG.Pand.1742100000000163
NL.IMBAG.Pand.1742100000000158
NL.IMBAG.Pand.1742100000000144
NL.IMBAG.Pand.1742100000000153
NL.IMBAG.Pand.1742100000000167

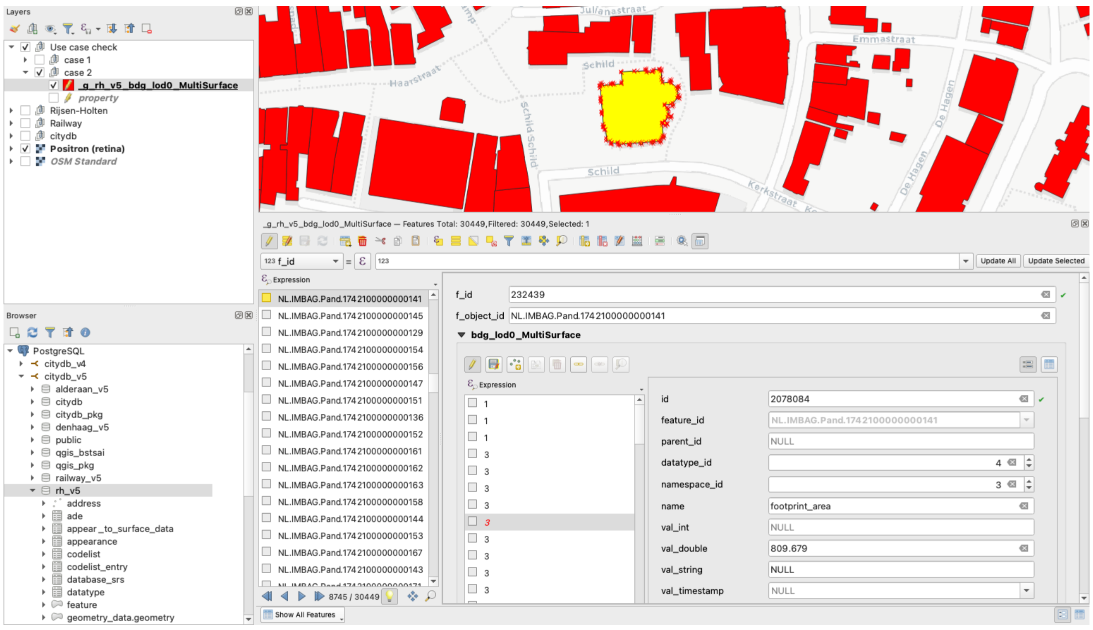
8745 / 30449

f_id	232439
f_object_id	NL.IMBAG.Pand.1742100000000141

bdg_lod0_MultiSurface

id	2078084
feature_id	NL.IMBAG.Pand.1742100000000141
parent_id	NULL
datatype_id	4
namespace_id	3
name	footprint_area
val_int	NULL
val_double	809.679
val_string	NULL
val_timestamp	NULL
val_uri	NULL

Figure 7.4.: Use case 2-1 read only checked - feature geometries viewed in 2D (Left) attribute table view. (Right) attribute form view



_g_rh_v5_bdg_lod0_MultiSurface — Features Total: 30449,Filtered: 30449,Selected: 1

f_id = 123

NL.IMBAG.Pand.1742100000000141
NL.IMBAG.Pand.1742100000000145
NL.IMBAG.Pand.1742100000000129
NL.IMBAG.Pand.1742100000000154
NL.IMBAG.Pand.1742100000000156
NL.IMBAG.Pand.1742100000000147
NL.IMBAG.Pand.1742100000000151
NL.IMBAG.Pand.1742100000000136
NL.IMBAG.Pand.1742100000000152
NL.IMBAG.Pand.1742100000000161
NL.IMBAG.Pand.1742100000000162
NL.IMBAG.Pand.1742100000000163
NL.IMBAG.Pand.1742100000000158
NL.IMBAG.Pand.1742100000000144
NL.IMBAG.Pand.1742100000000153
NL.IMBAG.Pand.1742100000000167
NL.IMBAG.Pand.1742100000000143

8745 / 30449

f_id	232439
f_object_id	NL.IMBAG.Pand.1742100000000141

bdg_lod0_MultiSurface

1
1
1
3
3
3
3
3
3
3
3
3

id	2078084
feature_id	NL.IMBAG.Pand.1742100000000141
parent_id	NULL
datatype_id	4
namespace_id	3
name	footprint_area
val_int	NULL
val_double	809.679
val_string	NULL
val_timestamp	NULL

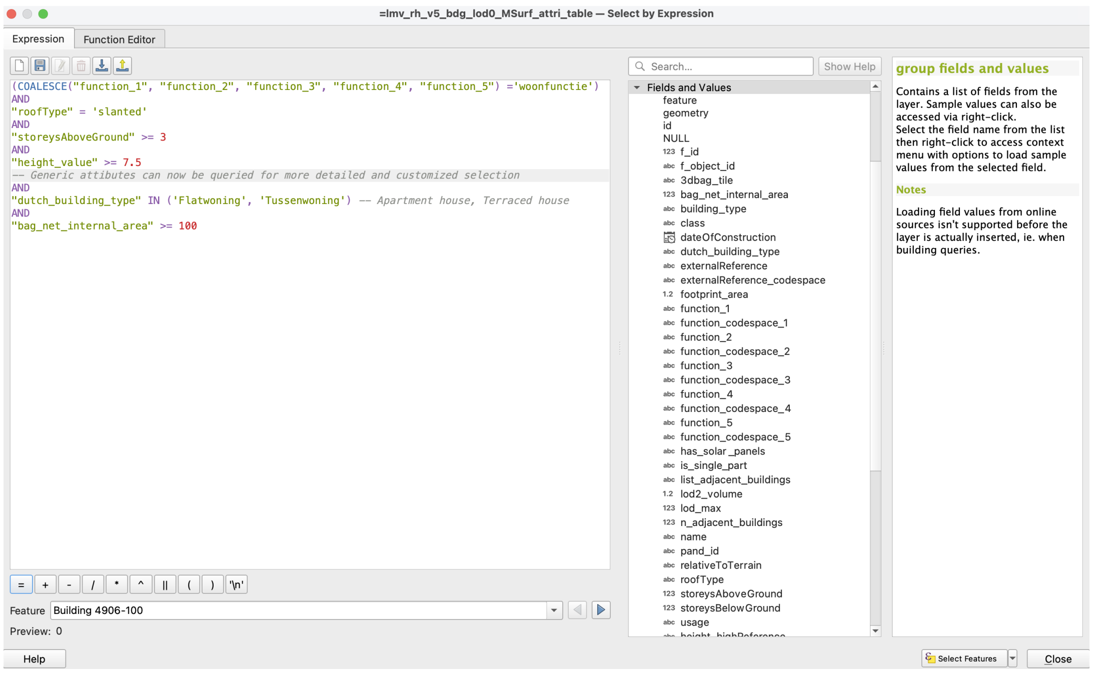
Figure 7.5.: Use case 2-2 read & write checked - attributes editing in attribute form view

7.3. Case 3: Users Interact with Visualised Feature Geometries and the "Linked" Attributes

Feature attributes stored within 3DCityDB v.5.0 can be classified into four attribute cases and flattened (linearised) using the corresponding approaches (see Section 4.3 and Section 5.4). Users can then specify their desired attributes and join them with the target feature geometries to create GIS layers. Each generated layer consists of a feature geometry materialized view and an integrated attribute table, which holds all user-selected attributes. The resulting join is stored as a materialized view for improved query performance (see Section 6.2).

The approach proposed in this research converts feature attributes stored in the cdb_schema PROPERTY table from the EAV model to the SFS model. In comparison to the use cases supported by the current plug-in for 3DCityDB v.4.x, where general and specific attributes are directly linked with geometries while generic attributes can only be accessed individually as sub-tables due to vertical database storage. In 3DCityDB v.5.0, generic attributes are flattened (linearised) and directly linked with geometries, providing direct access to all attributes. These linked attributes enhance the usability of CityGML data, as users can perform advanced queries via GIS layers associated with generic attributes, enabling more detailed analyses in QGIS (see Figure 7.6, Figure 7.7, and Figure 7.8, where attributes such as "dutch_building_type" and "bag_net_internal_area" become selectable in batch selection).

However, the approach proposed in this research only partially addresses use case 3 for users with access-only privileges, allowing them to view and inspect feature attributes without editing capabilities. Since the generated GIS layers are stored as materialized views, which are not automatically updatable like views in PostgreSQL (see Figure 7.9), use case 3 with read and write privileges is currently not achievable.


=lmv_rh_v5_bdg_lod0_MSurf_attri_table — Select by Expression

```
(COALESCE("function_1", "function_2", "function_3", "function_4", "function_5") ='woonfunctie')
AND
"roofType" = 'slanted'
AND
"storeysAboveGround" >= 3
AND
"height_value" >= 7.5
-- Generic attibutes can now be queried for more detailed and customized selection
AND
"dutch_building_type" IN ('Flatwoning', 'Tussenwoning') -- Apartment house, Terraced house
AND
"bag_net_internal_area" >= 100
```

Feature: Building 4906-100

Preview: 0

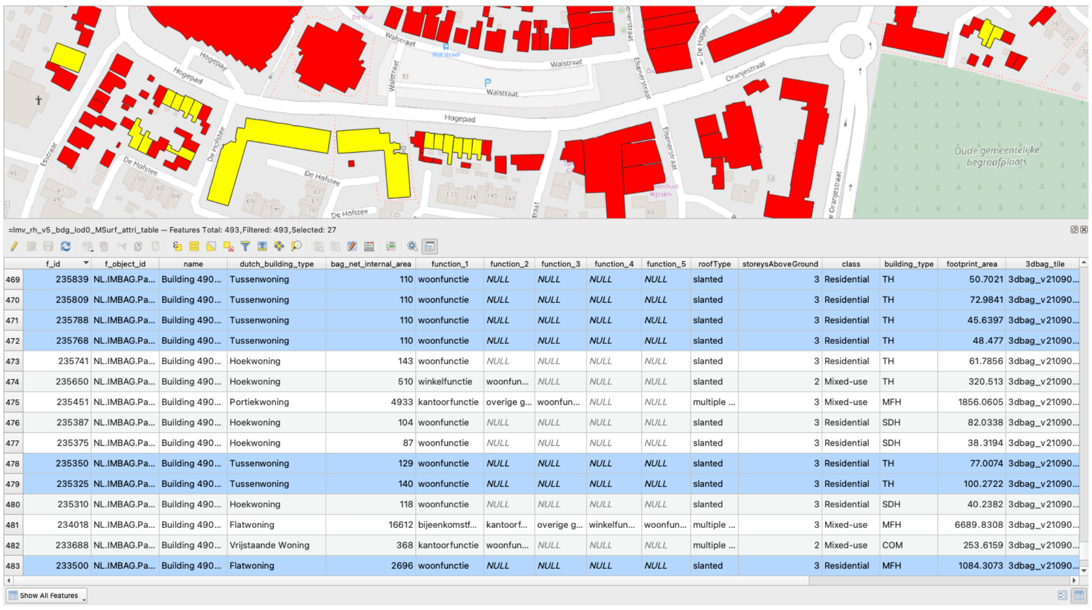
group fields and values

Contains a list of fields from the layer. Sample values can also be accessed via right-click. Select the field name from the list then right-click to access context menu with options to load sample values from the selected field.

Notes

Loading field values from online sources isn't supported before the layer is actually inserted, ie. when building queries.

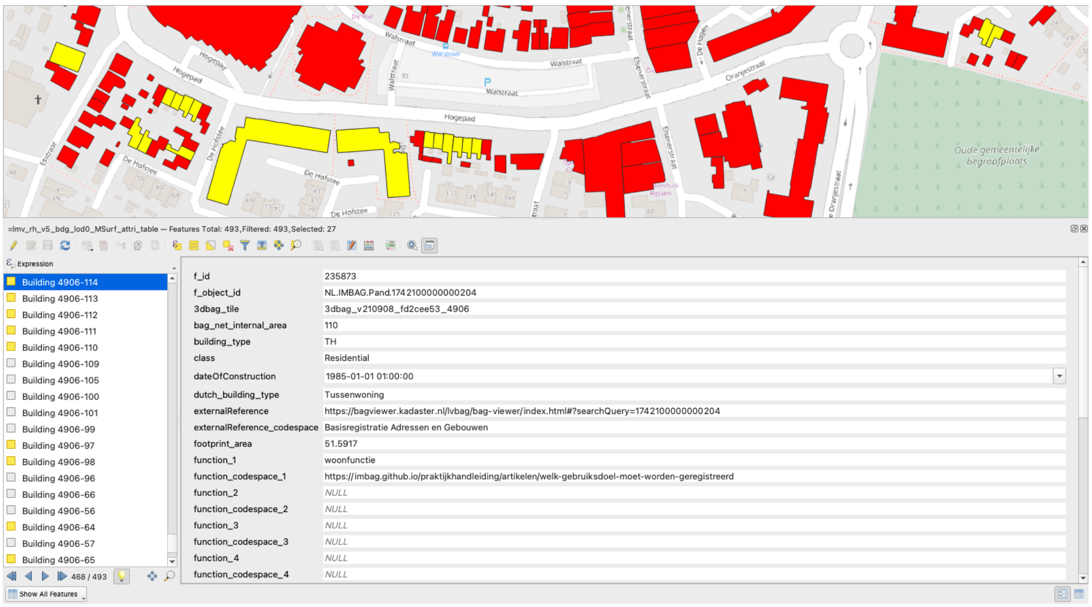
Figure 7.6.: Use case 3 checked - conditional query on flattened (linearised) feature attributes



=lmv_rh_v5_bdg_lod0_MSurf_attri_table — Features Total: 493, Filtered: 493, Selected: 27

	f_id	f_object_id	name	dutch_building_type	bag_net_internal_area	function_1	function_2	function_3	function_4	function_5	roofType	storeysAboveGround	class	building_type	footprint_area	3dbag_tile
469	235839	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	50.7021	3dbag_v21090...
470	235809	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	72.9841	3dbag_v21090...
471	235788	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	45.6397	3dbag_v21090...
472	235768	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	48.477	3dbag_v21090...
473	235741	NL.IMBAG.Pa...	Building 490...	Hoekwoning	143	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	61.7856	3dbag_v21090...
474	235650	NL.IMBAG.Pa...	Building 490...	Hoekwoning	510	winkelfunctie	woonfun...	NULL	NULL	NULL	slanted	2	Mixed-use	TH	320.513	3dbag_v21090...
475	235451	NL.IMBAG.Pa...	Building 490...	Portiekwoning	4933	kantoorfunctie	overige g...	woonfun...	NULL	NULL	multiple ...	3	Mixed-use	MFH	1856.0605	3dbag_v21090...
476	235387	NL.IMBAG.Pa...	Building 490...	Hoekwoning	104	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	SDH	82.0338	3dbag_v21090...
477	235375	NL.IMBAG.Pa...	Building 490...	Hoekwoning	87	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	SDH	38.3194	3dbag_v21090...
478	235350	NL.IMBAG.Pa...	Building 490...	Tussenwoning	129	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	77.0074	3dbag_v21090...
479	235325	NL.IMBAG.Pa...	Building 490...	Tussenwoning	140	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	100.2722	3dbag_v21090...
480	235310	NL.IMBAG.Pa...	Building 490...	Hoekwoning	118	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	SDH	40.2382	3dbag_v21090...
481	234018	NL.IMBAG.Pa...	Building 490...	Flatwoning	16612	bijeenkomstf...	kantoorf...	overige g...	winkelfun...	woonfun...	multiple ...	3	Mixed-use	MFH	6689.8308	3dbag_v21090...
482	233688	NL.IMBAG.Pa...	Building 490...	Vrijstaande Woning	368	kantoorfunctie	woonfun...	NULL	NULL	NULL	multiple ...	2	Mixed-use	COM	253.6159	3dbag_v21090...
483	233500	NL.IMBAG.Pa...	Building 490...	Flatwoning	2696	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	MFH	1084.3073	3dbag_v21090...

Figure 7.7.: Use case 3-1 read only checked - conditional query result (attribute table view)



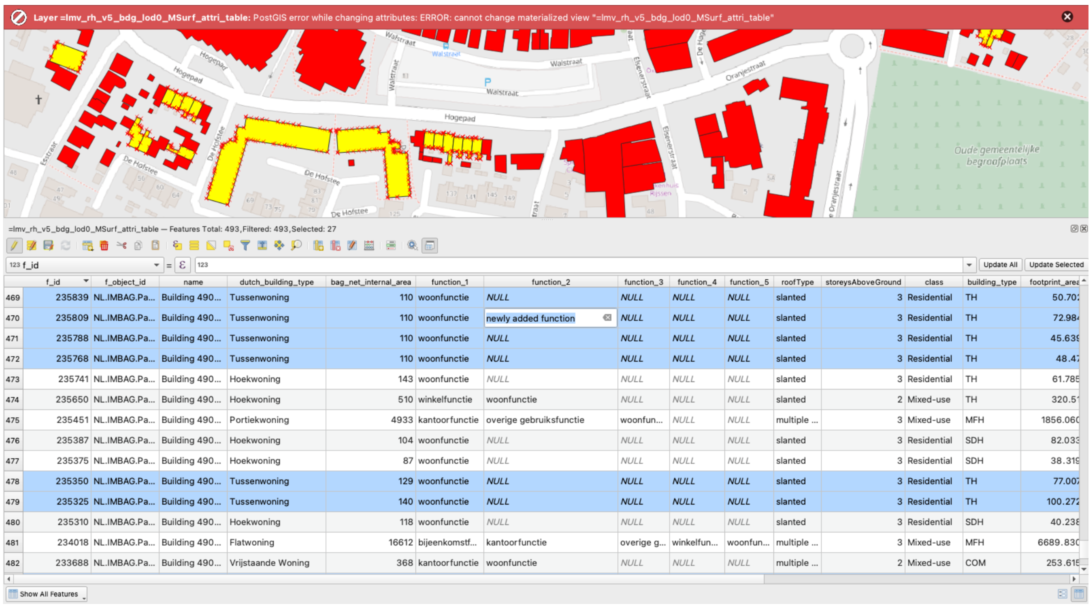
=lmv_rh_v5_bdg_lod0_MSurf_attri_table — Features Total: 493,Filtered: 493,Selected: 27

🟨 Building 4906-114	
🟨 Building 4906-113	
🟨 Building 4906-112	
🟨 Building 4906-111	
🟨 Building 4906-110	
☐ Building 4906-109	
☐ Building 4906-105	
☐ Building 4906-100	
☐ Building 4906-101	
☐ Building 4906-99	
🟨 Building 4906-97	
🟨 Building 4906-98	
☐ Building 4906-96	
☐ Building 4906-66	
☐ Building 4906-56	
🟨 Building 4906-64	
☐ Building 4906-57	
🟨 Building 4906-65	

468 / 493

f_id	235873
f_object_id	NL.IMBAG.Pand.1742100000000204
3dbag_tile	3dbag_v210908_fd2cee53_4906
bag_net_internal_area	110
building_type	TH
class	Residential
dateOfConstruction	1985-01-01 01:00:00
dutch_building_type	Tussenwoning
externalReference	https://bagviewer.kadaster.nl/lvbag/bag-viewer/index.html#?searchQuery=1742100000000204
externalReference_codespace	Basisregistratie Adressen en Gebouwen
footprint_area	51.5917
function_1	woonfunctie
function_codespace_1	https://imbag.github.io/praktijkhandleiding/artikelen/welk-gebruiksdoel-moet-worden-geregistreerd
function_2	NULL
function_codespace_2	NULL
function_3	NULL
function_codespace_3	NULL
function_4	NULL
function_codespace_4	NULL

Figure 7.8.: Use case 3-1 read only checked - conditional query result (attribute form view)



Layer =lmv_rh_v5_bdg_lod0_MSurf_attri_table: PostGIS error while changing attributes: ERROR: cannot change materialized view "=lmv_rh_v5_bdg_lod0_MSurf_attri_table"

=lmv_rh_v5_bdg_lod0_MSurf_attri_table — Features Total: 493,Filtered: 493,Selected: 27

123 f_id = 123

	f_id	f_object_id	name	dutch_building_type	bag_net_internal_area	function_1	function_2	function_3	function_4	function_5	roofType	storeysAboveGround	class	building_type	footprint_area
469	235839	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	50.70
470	235809	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	newly added function	NULL	NULL	NULL	slanted	3	Residential	TH	72.98
471	235788	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	45.63
472	235768	NL.IMBAG.Pa...	Building 490...	Tussenwoning	110	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	48.4
473	235741	NL.IMBAG.Pa...	Building 490...	Hoekwoning	143	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	61.78
474	235650	NL.IMBAG.Pa...	Building 490...	Hoekwoning	510	winkelfunctie	woonfunctie	NULL	NULL	NULL	slanted	2	Mixed-use	TH	320.5
475	235451	NL.IMBAG.Pa...	Building 490...	Portiekwoning	4933	kantoorfunctie	overige gebruiksfunctie	woonfun...	NULL	NULL	multiple ...	3	Mixed-use	MFH	1856.06
476	235387	NL.IMBAG.Pa...	Building 490...	Hoekwoning	104	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	SDH	82.03
477	235375	NL.IMBAG.Pa...	Building 490...	Hoekwoning	87	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	SDH	38.31
478	235350	NL.IMBAG.Pa...	Building 490...	Tussenwoning	129	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	77.00
479	235325	NL.IMBAG.Pa...	Building 490...	Tussenwoning	140	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	TH	100.27
480	235310	NL.IMBAG.Pa...	Building 490...	Hoekwoning	118	woonfunctie	NULL	NULL	NULL	NULL	slanted	3	Residential	SDH	40.23
481	234018	NL.IMBAG.Pa...	Building 490...	Flatwoning	16612	bijeenkomstf...	kantoorfunctie	overige g...	winkelfun...	woonfun...	multiple ...	3	Mixed-use	MFH	6689.83
482	233688	NL.IMBAG.Pa...	Building 490...	Vrijstaande Woning	368	kantoorfunctie	woonfunctie	NULL	NULL	NULL	multiple ...	2	Mixed-use	COM	253.61

Figure 7.9.: Use case 3-2 read and write checked - currently not available

7.4. Case 4: Users Perform Use Case 3 Using GUIs in QGIS

This research only focuses on adding server-side support to 3DCityDB v.5.0 in the plug-in. While the current implementation partially reproduces use case 3, allowing users to interact with feature geometries and the flattened (linearised) attributes in QGIS with read-only privileges, it does not fully address the interactive editing capabilities that users may require.

Therefore, use case 4 should be considered for further development on the client-side of the plug-in, and future work should continue enhancing the plug-in to enable full read and write access directly through GUIs in QGIS. By implementing this capability, users would be able to perform real-time edits on GIS layers, with changes reflected in the underlying database.

8. Conclusion and Future Development

This chapter concludes this thesis in the research overview and answers the research questions in Section 8.1. This thesis's main contributions and limitations are discussed in Section 8.2, followed by the research recommendations for future development.

8.1. Research Overview

The 3DCityDB-Tools plug-in for QGIS (plug-in) developed by the 3D Geoinformation team at TU Delft facilitates the management and visualisation of spatial data stored within the 3DCityDB, which currently supports CityGML v.1.0, v.2.0 and CityJSON [1]. This thesis aims to enhance the plug-in by developing server-side support to create GIS layers from spatial data encoded in 3DCityDB v.5.0. This new open-source geo-relational database adheres to CityGML v.3.0 standards. It will extend the capabilities of plug-in.

The research begins with an overview of the significant changes introduced by CityGML v.3.0 and the corresponding differences in 3DCityDB encoding. This includes new concepts of LoD and geometry, updated 3DCityDB schema tables, and the EAV model storage of all feature attributes. By identifying potential use cases for CityGML data in QGIS via the plug-in, the goal becomes clear: to create layers comprising feature geometries associated with flattened (linearised) attributes for user interaction with 3DCityDB-encoded data in QGIS.

To create these GIS layers, this thesis presents the concept of feature geometry and attribute metadata checks concerning the encoding characteristics of 3DCityDB v.5.0. Feature geometries are categorised into Space, Boundary, Relief, and Relief Component features, each with a custom query template for collection in geometry views. Feature attributes are classified into Inline-Single, Inline-Multiple, Nested-Single, and Nested-Multiple classes, with specific methods for collecting and flattening them into attribute views. By storing the combined target feature geometry and attribute tables in materialized views, users can access various feature LoDs and geometry representations, along with the existing flattened (linearized) attributes. This allows users to explore and utilise CityGML data encoded in 3DCityDB v.5.0 with read-only access by importing the generated GIS layers into QGIS, similar to the current capabilities of the plug-in. The PL/pgSQL functions developed in this thesis for creating GIS layers from data in 3DCityDB v.5.0 are available at the GitHub repository: `https://github.com/bsttsai/3DCityDB-Tools-for-QGIS_beta/tree/thesis_final`.

To reproduce the capabilities of the current plug-in that allows users to interact with the CityGML data encoded with 3DCityDB v.5.0 via GIS layers, the main research question is determined as:

How does the new database structure of 3DCityDB v.5.0 affect the current methods of the plugin to create layers containing both geometries and attributes for a selected feature type following the SFS model?

The following paragraphs define and answer four sub-questions to answer this question.

1. **How do the new CityGML v.3.0 concepts of space and LoD affect the process?**
   In CityGML v.3.0, the concepts of LoD and geometry have been elevated from the thematic module level to the core module level of the data model. In CityGML v.2.0, each class specified its possible LoDs geometry representations. The current plug-in traverses existing classes within a user-specified module to generate all possible feature geometry representations according to these specifications. Feature geometries are decomposed into polygons within the 3DCityDB v.4.x encoding, and relevant polygons for a specified feature geometry representation are collected and aggregated when accessed.

   In contrast, CityGML v.3.0 derives the spatial concepts of all features from four top-level classes: ”AbstractSpace”, ”AbstractThematicSurface”, ”ReliefFeature”, and ”AbstractReliefComponent”. These four abstract classes categorise features into Space, Boundary, Relief, and Relief Component features, each with a limited set of possible LoDs geometry representations. This structural change allows for a schema-wide scan to check for the existence of feature geometries, after which users can select their desired feature geometry representation and LoD for visualisation in QGIS via materialized views. The shift in spatial concepts in CityGML v.3.0 enhances the flexibility of the methods used to create feature geometry views, which form the foundation of GIS layers.

2. **Regarding geometries, can the same or a similar approach be reproduced?**
   This question is addressed in Section 4.2.2. Below is a summary of the conclusions on creating feature geometry views:

   - **Is it still necessary to rely on materialized views?**
     Relying on materialized views is not strictly necessary, particularly for relief component features and space features without implicit representations, as the geometry roots are directly stored in the 3DCityDB v.5.0 encoding. However, materialized views remain preferable when working with large datasets, as they offer faster access to the queried results.

   - **What alternatives are available?**
     For space and relief component features with geometry properties, PostgreSQL views can be used to store the collected feature geometries. The difference in query time between these two types of views is minimal.

3. **Regarding attributes, can the same or a similar approach be reproduced?**
   This question is discussed in Section 4.3.2, Section 5.4.4, and Section 6.2. Below is a summary of the conclusions on creating feature attribute views:

   - **Is it still necessary to rely on updatable views?**
     Since all feature attributes are stored following EAV model in 3DCityDB v.5.0, flattening these attributes is necessary when accessing them via GIS layers. However, updatable views in PostgreSQL are not the optimal solution for storing the flattened (linearised) attribute results, as re-computation during attribute flattening can be complex and time-consuming.

   - **What alternatives are available?**
     For better query performance, materialized views are preferable when accessing feature attributes through GIS layers. In future work, certain trigger functions should be developed to reflect user changes in QGIS and update the underlying source tables accordingly.

4. **How is the CityGML v.2.0 data mapped to the new schema of 3DCityDB v.5.0?**
   The mapping of CityGML v.2.0 data to the new 3DCityDB v.5.0 schema is based on the identifiers of each class, adhering to the namespace standards of 3DCityDB v.5.0 encoding, which is still under development. The mapping rules for CityGML classes are stored as JSON texts in the `schema` column within the 3DCityDB OBJECTCLASS table. For example, the mapping rules for the ”Building” class are shown in Listing 8.1. The identifier specifies the table used for mapping the class, and properties are encoded according to the `type` derived from the rules specified in the `namespace`, which references the 3DCityDB NAMESPACE table.

```json
{
     "identifier" : "bldg:Building",
     "table" : "feature",
     "properties" : [
         {
             "name": "buildingPart",
              "namespace" : "http://3dcitydb.org/3dcitydb/building/5.0",
             "type": "core:FeatureProperty"
         }
     ]
}
```

Listing 8.1: Mapping example in JSON - ”Building” Class

- **Can we handle CityGML v.2.0 data as CityGML v.3.0 data if it is stored in 3DCityDB v.5.0?**
  By following the mapping rules specified in the `schema` column of the 3DCityDB OBJECTCLASS table, CityGML v.2.0 data is automatically mapped to CityGML v.3.0 and stored accordingly within 3DCityDB v.5.0, with some exceptions based on CityGML v.3.0 standards, allowing it to be processed as CityGML v.3.0 data. However, certain properties from CityGML v.2.0 cannot be directly mapped to CityGML v.3.0. For instance, the ”roof edge” geometry type, which has been removed in CityGML v.3.0, is marked as ”deprecated” according to the 3DCityDB NAMESPACE table. The PL/pgSQL functions developed in this research still display these deprecated properties in the feature metadata check but exclude them during view creation.

8.2. Discussion

8.2.1. Contributions

The `qgis_pkg` developed in this research successfully demonstrates the capability to reproduce the GIS layer creation process from CityGML data stored within the new 3DCityDB v.5.0 encoding, marking an advancement in the integration and utilisation of spatial data within the updated database framework. Compared to the current plug-in, the added support for 3DCityDB v.5.0 on the server side enhances user flexibility in interacting with 3DCityDB-encoded data, particularly in terms of feature geometry and attribute management:

- **Feature Geometries**
  Users now have the option to store collected feature geometries in either views or materialized views. The `create_geometry_view` function allows users to specify the desired view type in the input parameters. Views can be used to visualise feature geometries in QGIS when working with relatively small datasets, offering a lighter and more responsive option, particularly when changes are made to the source tables.

  For checking target feature geometries, users can specify the LoDs and geometry representations for each class within a CityGML module when creating views rather than generating views for all classes simultaneously. The selectable feature geometry view creation could shorten the view generation time, providing a better user experience.

- **Feature Attributes**
  Feature attributes become selectable for the users when creating GIS layers. The `create_attribute_view` function enables individual attribute view creation, and the `create_attris_table_view` function allows users to specify their desired attribute and integrate them into an attribute table, which is updated when the attributes of the corresponding class are selected.

  Additionally, the general, specific, and generic feature attributes are all flattened by the implementation of this thesis. Unlike the current plug-in, where generic attributes can only be inspected as sub-tables associated with the GIS layers in the ”attribute form” view in QGIS, users can now inspect attributes in the ”attribute table” view as well. This enhancement improves the usability of CityGML data, as generic attributes can be batch-queried using the analysis tools in QGIS.

Overall, this research contributes substantially to the usability and adaptability of spatial data management within the updated 3DCityDB v.5.0 framework, providing a more tailored and flexible user experience in QGIS.

8.2.2. Limitations

Although this thesis successfully demonstrates the creation of GIS layers from data stored within the new 3DCityDB v.5.0 schema, certain limitations still exist. These can be summarised as follows:

- **Read-only privileges in Use Case 3**: A key limitation is that the generated GIS layers can only be visualised and inspected in QGIS. Since materialized views in PostgreSQL are not as updatable as regular views, any changes made to the source tables require manual refreshes of the layers to keep them up-to-date. Additionally, editing feature attributes via these layers in QGIS is not yet possible. Enabling read and write privileges for Use Case 3 would require additional functions, such as triggers [47], to handle insert, update, and delete operations. Future work should explore managing user-initiated changes within QGIS to provide full support for 3DCityDB v.5.0 on the plug-in server side.
- **Features without direct LoDs and geometry representations cannot be viewed via GIS layers**: As indicated in the layer generation results of transportation datasets (Section 6.4), certain features and their attributes cannot be viewed and checked via the GIS layers. For example, the features of the ”TrafficSpace” and ”AuxiliaryTrafficSpace” classes in the CityGML v.3.0 ”Transportation” module can have granularity attributes as lane or way [20]. However, only their child boundary features, (auxiliary) traffic

areas, have direct LoDs and geometry representations in the Munich and New York City transportation datasets. This results in the missing geometry views for joining attributes like the granularity to form visible layers.

Furthermore, the Munich and New York City transportation datasets also contain features such as roads, sections, intersections and squares. They are structured as nested features to express the semantic hierarchy of Road-Section/Intersection-(Auxiliary) Traffic space-(Auxiliary) Traffic area; however, they do not possess direct geometry representation for GIS layer visualisation and can only be viewed by gathering their child feature geometries, i.e. (auxiliary) traffic areas. Since modules like "Transportation" in CityGML v.3.0 are modified to incorporate more detailed feature types, it requires further investigation to view the features without direct spatial properties and access their attributes via GIS layers.

- **Limited testing with CityGML v.3.0 datasets**: The `qgis_pkg` developed in this research has yet to be thoroughly tested due to the limited availability of CityGML v.3.0 data. Conducting such tests is crucial to validate the proposed method for GIS layer creation and to ensure full compatibility with 3DCityDB v.5.0 for the plug-in.

8.2.3. Future Development

As discussed in the possible use case evaluation (Chapter 7) and the limitations (Section 8.2.2), the following paragraphs outline the directions for further development aimed at providing full support for 3DCityDB v.5.0 within the plug-in.

1. **Optimisation of the Approach for GIS Layer Creation**
   Future development of the plug-in should focus on optimising the approach for creating GIS layers to reflect user changes in the source tables within QGIS. Key areas for improvement include:

   a) **Dynamic Updates**
      Explore methods to ensure that GIS layers are dynamically updated in real-time, reflecting user changes in QGIS. This may involve implementing trigger functions to handle insert, update, and delete operations within the database. Additionally, the concept of Incremental View Maintenance (IVM) could be referenced as a potential approach to make materialized views updatable, ensuring they remain current without full recomputation.

      IVM keeps materialized views up-to-date by calculating and applying only the changes rather than fully re-computing the view as the `REFRESH MATERIALIZED VIEW` command does. This is more efficient, especially when only small parts of the view are modified [48]. Materialized views that support IVM are called Incrementally Maintainable Materialized View (IMMV), enabled by the `pg_ivm` extension in PostgreSQL [49]. The `create_immv` function allows users to create IMMVs with a relation name and a view definition query. When an IMMV is created, triggers are automatically set up to enable immediate updates to the view in response to modifications. Unique indices are also automatically created on primary key columns, `GROUP BY` expressions, and columns with a `DISTINCT` clause. IMMVs provides faster updates of materialized views compared to the normal refresh command at the cost of slower base table updates since triggers process each modification.

Although IMMVs provide a solution for updatable materialized views, they cannot be directly applied to this research due to two main reasons:

i. IMMVs created by the `pg_ivm` extension only supports simple view definitions and does not allow materialized views defined with `OUTER JOIN`s. As the proposed approach involves flattening and joining user-selected attributes into an integrated attribute table using `OUTER JOIN`s, it does not meet IMMV requirements.

ii. Changes made in QGIS are updated to the underlying 3DCityDB schema via the generated GIS layers. This reversed approach is not supported by IMMVs. Moreover, IVM may be slower than normal refreshes when base tables are frequently modified. This suggests that relying on IMMVs for batch-editing updates in QGIS may not be ideal, even if reversed updates were possible.

Despite the limitations of IMMVs, the underlying concepts of IVM still offer valuable insights into updating materialised views. When enabling users to read and write to the generated GIS layers, these ideas should be considered.

b) **Performance and Scalability**
Enhancing the performance of materialized views and other database operations involves optimising query structures and refresh strategies. The goal is to ensure the system can handle large datasets and frequent updates efficiently without compromising performance.

The approach used in this research generates a GIS layer by creating a materialized view that consists of a feature geometry joined with an integrated attribute table. The content of this attribute table is based on the user's selection of attributes. These selected attributes are flattened (linearized) to follow the SFS model while they are stored in the source table using the EAV model. Therefore, any changes made to a layer within QGIS would require corresponding trigger functions to detect these events and convert the changed attributes from the SFS model back into the EAV model. This process is complex and challenging, as it would require dynamically creating numerous trigger functions based on the selected attributes, leading to significant implementation and maintenance overhead.

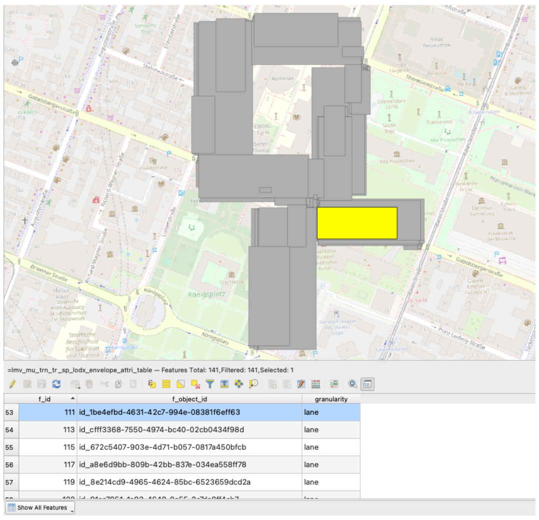
To simplify the implementation of GIS layer updates, an alternative approach could involve duplicating the data of the generated GIS layer into a ”flattened” table that holds the modifications performed by the users. The changes to the ”flattened” table can then be updated back to the underlying `cdb_schema` PROPERTY table, reducing the workload in handling user updates in QGIS. However, this alternative requires further verification and should be investigated in future development of the plug-in on the server side.

2. **Access the features without direct spatial properties via GIS layers**
Two possible solutions are proposed for users to view the features without direct geometry representations and access their attributes via GIS layers (see Section 8.2.2):

a) **Feature bounding box envelopes**
Using the feature envelopes as the alternative geometries to join the attributes could be the simplest and fastest approach to creating layers for visualisation. The feature envelopes are directly accessible in the FEATURE table without joining the GEOMETRY_DATA table. The selected envelopes can then be saved in materialized views named with LoDx and envelope, indicating that they are not

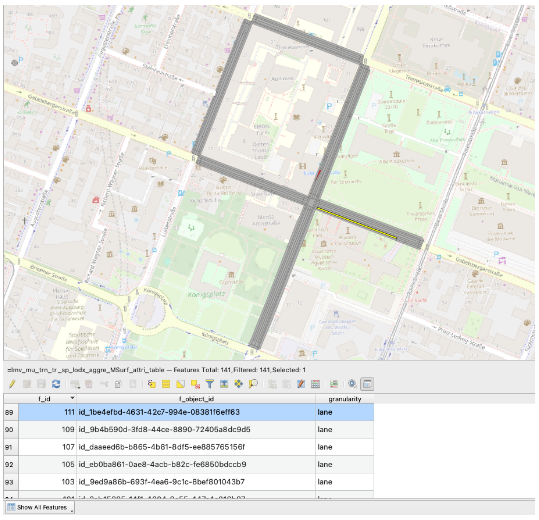
the existing feature geometry representation of the target features. Figure 8.1a shows the layer created using (auxiliary) traffic space envelopes from the Munich transportation dataset. It allows users to access the granularity values, giving semantic information about the free spaces above the traffic area used as lanes. However, the bounding box envelopes of the features are not suitable for checking the feature spatial properties due to the overlapping geometries and the coarse representation.

b) **Aggregation of child feature geometries**
Aggregating the child feature geometries as the alternative geometries to join the attributes is the other approach for creating layers. In the Munich transportation dataset, the (auxiliary) traffic spaces comprise (auxiliary) traffic areas. It is then reasonable to collect corresponding child boundary features and aggregate the geometries to represent their parent space features. Figure 8.1b shows the layer created using the aggregated (auxiliary) traffic area geometries, which can be a possible solution for users to check the ”geometry-less” attributes.


=lmv_mu_trn_tr_sp_lodx_envelope_attri_table — Features Total: 141,Filtered: 141,Selected: 1

	f_id	f_object_id	granularity
53	111	id_1be4efbd-4631-42c7-994e-08381f6eff63	lane
54	113	id_cfff3368-7550-4974-bc40-02cb0434f98d	lane
55	115	id_672c5407-903e-4d71-b057-0817a450bfcb	lane
56	117	id_a8e6d9bb-809b-42bb-837e-034ea558ff78	lane
57	119	id_8e214cd9-4965-4624-85bc-6523659dcd2a	lane

(a) Envelope feature geometries

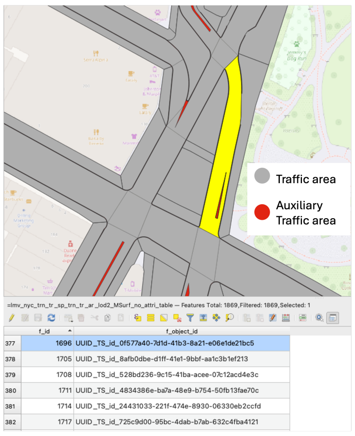

=lmv_mu_trn_tr_sp_lodx_aggre_MSurf_attri_table — Features Total: 141,Filtered: 141,Selected: 1

	f_id	f_object_id	granularity
89	111	id_1be4efbd-4631-42c7-994e-08381f6eff63	lane
90	109	id_9b4b590d-3fd8-44ce-8890-72405a8dc9d5	lane
91	107	id_daaeed6b-b865-4b81-8df5-ee885765156f	lane
92	105	id_eb0ba861-0ae8-4acb-b82c-fe6850bdccb9	lane
93	103	id_9ed9a86b-693f-4ea6-9c1c-8bef801043b7	lane

(b) Aggregated child boundary feature geometries

Figure 8.1.: Alternative geometry options to generate layers for the (auxiliary) traffic spaces (Munich transportation dataset)

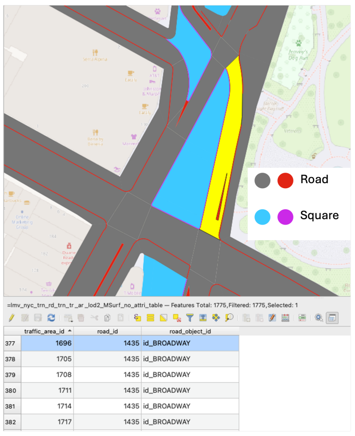
Child feature geometries aggregation could be a more suitable approach for viewing parent features as layers, especially to check the semantic hierarchy. This could involve using more advanced SQL queries to integrate semantic information into features with direct geometry representations, such as the (auxiliary) traffic areas in the Munich and New York City transportation datasets (Figure 8.2a), and then storing the results as additional layers. This approach allows users to visualise roads and squares (Figure 8.2b) or more detailed semantic elements like road sections and intersections (Figure 8.2c). Further discussion and investigation are required to develop the solutions on the plug-in server side to provide comprehensive support for CityGML v.3.0 data applications.



=lmv_nyc_trn_tr _sp_trn_tr _ar _lod2_MSurf_no_attri_table — Features Total: 1869,Filtered: 1869,Selected: 1

	f_id	f_object_id
377	1696	UUID _TS_id_0f577a40-7d1d-41b3-8a21-e06e1de21bc5
378	1705	UUID _TS_id_8afb0dbe-d1ff-41e1-9bbf-aa1c3b1ef213
379	1708	UUID _TS_id_528bd236-9c15-41ba-acee-07c12acd4e3c
380	1711	UUID _TS_id_4834386e-ba7a-48e9-b754-50fb13fae70c
381	1714	UUID _TS_id_24431033-221f-474e-8930-06330eb2ccfd
382	1717	UUID _TS_id_725c9d00-95bc-4dab-b7ab-632c4fba4121

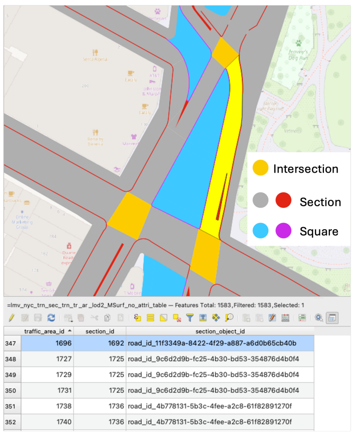
(a) Original layer of (Auxiliary) traffic areas



=lmv_nyc_trn_rd_trn_tr _ar _lod2_MSurf_no_attri_table — Features Total: 1775,Filtered: 1775,Selected: 1

	traffic_area_id	road_id	road_object_id
377	1696	1435	id_BROADWAY
378	1705	1435	id_BROADWAY
379	1708	1435	id_BROADWAY
380	1711	1435	id_BROADWAY
381	1714	1435	id_BROADWAY
382	1717	1435	id_BROADWAY

(b) Additional semantic layer of Road & Square



=lmv_nyc_trn_sec_trn_tr _ar _lod2_MSurf_no_attri_table — Features Total: 1583,Filtered: 1583,Selected: 1

	traffic_area_id	section_id	section_object_id
347	1696	1692	road_id_11f3349a-8422-4f29-a887-a6d0b65cb40b
348	1727	1725	road_id_9c6d2d9b-fc25-4b30-bd53-354876d4b0f4
349	1729	1725	road_id_9c6d2d9b-fc25-4b30-bd53-354876d4b0f4
350	1731	1725	road_id_9c6d2d9b-fc25-4b30-bd53-354876d4b0f4
351	1738	1736	road_id_4b778131-5b3c-4fee-a2c8-61f82891270f
352	1740	1736	road_id_4b778131-5b3c-4fee-a2c8-61f82891270f

(c) Additional semantic layer of (Inter)Section & Square

Figure 8.2.: Create additional semantic layers (New York City transportation dataset)

3. **Client-Side Adaptations for 3DCityDB v.5.0 in the plug-in GUIs**
   After updating the plug-in server-side to support 3DCityDB v.5.0, necessary client-side adaptations should be made to ensure seamless interaction and enhanced usability. Key areas for adaptation include:

   a) **"Layer Loader" Adaptation for Selectable Feature LoDs, Geometry Representations, and Attributes**
      The plug-in GUIs should be enhanced to allow users to select their desired feature LoD and geometry representation when creating geometry views. Additionally, users should be able to choose which feature attributes to include in GIS layers. This involves:

      i. Implementing a user-friendly interface for displaying the available LoDs and geometry representations from the geometry metadata table.

      ii. Providing options to filter and select specific attributes for display in the GIS layers, with options to include or exclude attributes based on relevance from the attribute metadata table.

      iii. Updating the GUI to support these selections dynamically, ensuring that changes are reflected immediately in the layer creation process.

   b) **"Bulk Deleter" Adaptation Regarding the Feature Editing Approach of the GIS Layers**
      Adapt the "Bulk Deleter" function to align with the new feature editing capabilities and workflows in 3DCityDB v.5.0. This includes:

      i. Enhancing the interface to support bulk deletion of features directly from the GIS layers, incorporating user feedback mechanisms to confirm and manage deletions.

      ii. Integrating new functionalities that respect the concurrent editing constraints and updating the underlying data accordingly, possibly by utilising trigger functions or synchronisation methods to ensure consistency.

c) **User Experience Enhancements**: Offering customisable settings for users to save their preferences for feature LoD, geometry representations, and attributes. Streamlining repeated tasks, such as setting up optional toggle buttons for joining all available attributes regarding a target class to improve workflow efficiency.

8.2.4. Reflection and Outlook

The core contribution of this thesis is the linearisation of ”complex features”, which are the ”nested attributes” referred to by this research in CityGML data encoded within 3DCityDB, making them compatible with SFS model and more accessible for users in QGIS. This approach aims to bridge the gap between complex, multi-tiered data schemas and more simplified, usable formats that can be easily viewed and queried.

While the WFS provides a convenient method for creating, modifying, and exchanging CityGML data in QGIS over the Internet, challenges persist in handling complex feature schemas, especially since all GIS software relies upon data structured following SFS model. The QGIS WFS provider was previously limited to consuming features returned as GML simple features, referred to as ”inline attributes” in this research. To access complex features via WFS, a workaround involving the QGIS GML Application Schema Toolbox plugin (GMLAS) plug-in is used [50]. The GMLAS can consume complex features either by linking the initial XML hierarchical view to GIS layers or by converting the data to a relational database. In the latter case, the data is spread across different GIS layers, and the relationships between tables are defined in QGIS using relation reference widgets, enabling navigation through the standard QGIS attribute table in the ”forms view”. However, working with complex features using GMLAS requires users to be aware of its limitations, and it does not offer the same level of user experience as working with simple feature WFS layers [51].

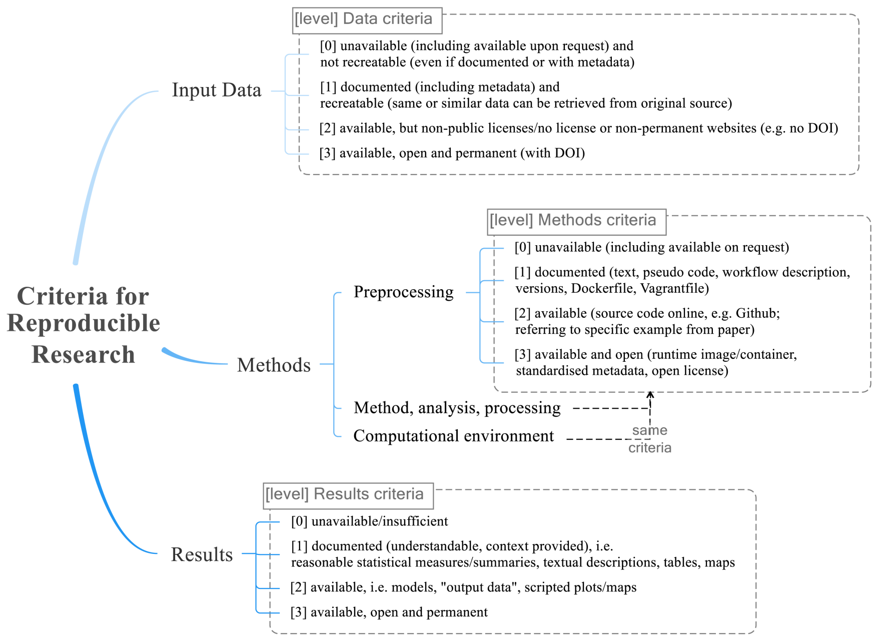
For supporting complex features consumption in WFS providers, QGIS was recently enhanced with a solution proposed by the QGIS-DE user group to expose complex feature properties as JSON content converted from XML [51]. The proposed solution aims to enhance QGIS’s capability to handle complex feature schemas in WFS providers by flattening nested XML structures as JSON serialised strings, which are compatible with the existing infrastructure of WFS providers. This conversion allows users to view and interact with complex GML data in QGIS attribute tables. However, the implementation is still in development, with limitations, such as supporting only one geometry field per WFS layer. This implies that if a fetched WFS layer has multiple geometry representations, only one will be accessible at a time in QGIS. Additionally, WFS with Transactions (WFS-T) is not supported, meaning users are limited in editing capabilities for writing changes back to WFS providers. Finally, server-side filtering is restricted to non-XML fields, so only simple feature properties can be queried.

As highlighted in ongoing developments within the QGIS community, flattening XML data served from WFS is still an evolving process. QGIS WFS providers are yet to support complex feature structures in a user-friendly manner fully, limiting access and editing of CityGML data in real-time. In contrast, the approach proposed by this research offers an alternative. Except for handling complex features on the client side as the WFS providers offer, the research approach flattens them on the server side in the database. By downloading CityGML data and encoding it with 3DCityDB, users can linearise nested features and complex attributes to access them as usual GIS layers since they are flattened to follow SFS model. This conversion simplifies the structure, providing direct access to feature attributes via QGIS’s attribute table, making the dataset more accessible for analysis and manipulation.

Looking to the future, if users are granted read-and-write privileges (use case 3-2) to the linearised attribute tables, it will enable the potential for users to edit attributes directly in QGIS. Any changes can then be pushed back to the underlying 3DCityDB schema tables and exported to CityGML files via the 3DCityDB for broader data exchange purposes. While editing CityGML attributes directly through 3DCityDB might require more effort, flattening (linearising) the feature attributes encoded within 3DCityDB offers a viable solution for users to interact with and use the rich CityGML data. The proposed approach of this thesis aligns with ongoing development efforts surrounding complex features in WFS providers, offering a complementary pathway for handling complex geospatial data.

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
[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)
Method, analysis, processing
Computational environment
same criteria
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.

Grade/evaluate yourself for the five criteria (giving 0/1/2/3 for each):

1. **Input Data:** The CityGML datasets are available, open, and permanent on websites. However, only a portion of the test datasets are provided due to GitHub's file size limitation, which slightly reduces accessibility. **[2]**
2. **Preprocessing:** The CityGML datasets can be imported using the open-source 3DCityDB command line tool, making it a straightforward process. **[3]**
3. **Methods:** The proposed PL/pgSQL functions are installable and applicable across multiple 3DCityDB v.5.0 instances, offering flexibility and reproducibility for creating GIS layers.**[3]**
4. **Computational Environment:** pgAdmin4, QGIS, and 3DCityDB v.5.0 are all open-source tools and can be freely set up, which makes the computational environment highly accessible and reproducible. **[3]**

5. **Results:** The PL/pgSQL functions are available on GitHub, enabling others to reproduce the results using the listed tools. Although the dataset limitation slightly affects input availability, the method and environment are reproducible. **[3]**

A.2. Self-reflection

This thesis presents an experimental approach to creating GIS layers from CityGML data stored within the new 3DCityDB v.5.0, partially replicating the capabilities of the existing plug-in. The CityGML test datasets are primarily accessible via online data portals, and users can generate GIS layers after setting up the required open-source applications.

The results demonstrate a more flexible method for interacting with 3DCityDB-encoded spatial data, allowing users to select desired feature LoD geometry representations and attributes for visualisation via GIS layers. The most significant contribution is the successful flattening (linearisation) of ”nested attributes.” Handling the complex features of GML schemas in QGIS remains a challenge, even with XML-to-JSON conversion support in WFS layers. While the current approach only allows users to inspect feature attributes with read-only privileges, it shows potential for future development, particularly in enabling editability. This offers a promising pathway to more comprehensive CityGML data usage.

On a personal level, this thesis has deepened my understanding of the CityGML data model and the capabilities of the open-source 3DCityDB. Although fixing bugs in the functions was sometimes frustrating, I found it rewarding to explore new solutions and refine my understanding of the detailed CityGML standards through trial and error.

Bibliography

[1] Giorgio Agugiaro, Konstantinos Pantelios, Tendai Mbwanda, and Camilo León Sánchez. 3DCityDB-Tools-for-QGIS. https://github.com/tudelft3d/3DCityDB-Tools-for-QGIS, 2024.

[2] Filip Biljecki, Jantien Stoter, Hugo Ledoux, and Zlatanova. Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information*, 4(4): 2842–2889, 2015. ISSN 2220-9964. doi: 10.3390/ijgi4042842. URL https://www.mdpi.com/2220-9964/4/4/2842.

[3] Gerhard Gröger, Thomas H Kolbe, Claus Nagel, and Karl-Heinz Häfele. OGC City Geography Markup Language (CityGML) Encoding Standard. Open Geospatial Consortium, 2012. https://portal.ogc.org/files/?artifact_id=47842.

[4] W3C. XML Path Language (XPath) 2.0. https://www.w3.org/TR/xpath20/, 2010. Accessed on: 2024-05-16.

[5] Ledoux. CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(1):4, Jun 2019. ISSN 2363-7501. doi: 10.1186/s40965-019-0064-0. URL https://doi.org/10.1186/s40965-019-0064-0.

[6] Oracle. What Is a Database? https://www.oracle.com/database/what-is-database/, 2024. Accessed on: 2024-08-24.

[7] Oracle. What is a Relational Database (RDBMS). https://www.oracle.com/database/what-is-a-relational-database/, 2024. Accessed on: 2024-08-24.

[8] PostgreSQL. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/, 2024. Accessed on: 2024-08-29.

[9] Oracle. Oracle database. https://www.oracle.com/database/, 2024. Accessed on: 2024-08-29.

[10] PostGIS. Postgis: Spatial and geographic objects for postgresql. https://postgis.net/, 2024. Accessed on: 2024-09-03.

[11] Hugo Ledoux and Gina Stavropoulou and Leon Powałka and Yuduan Cai and Siebren Meines and Chris Poon and Yitong Xia and Lan Yan. cjdb. https://github.com/cityjson/cjdb, 2024. Accessed on: 2024-09-09.

[12] 3DCityDB(2023). 3DCityDB, 3D City Database Documentation, 2023. https://3dcitydb-docs.readthedocs.io. Accessed on: 2024-05-16.

[13] Zhihang Yao, Claus Nagel, Felix Kunde, Hudra, Andreas Donaubauer, Thomas Adolphi, and Thomas 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):1–26, 2018.

[14] 3DCityDB(Suite). 3DCityDB Suite. https://github.com/3dcitydb/3dcitydb-suite, 2023. Accessed on: 2024-05-13.

[15] Giorgio Agugiaro, Konstantinos Pantelios, Camilo León-Sánchez, Zhihang Yao, and Claus Nagel. Introducing the 3dcitydb-tools plug-in for qgis. In Thomas H. Kolbe, Andreas Donaubauer, and Christof Beil, editors, *Recent Advances in 3D Geoinformation Science*, Lecture Notes in Geoinformation and Cartography, pages 797–821. Springer, 2024. ISBN 978-3-031-43698-7. doi: 10.1007/978-3-031-43699-4_48. URL https://www.3dgeoinfo.org/3dgeoinfo/.

[16] Nintex. XML Data Types. https://help.nintex.com/en-US/k2blackpearl/userguide/4.6.10/XML_Data_Types.html, 2024. Accessed on: 2024-10-21.

[17] QGIS Documentation Team. Simple Feature Model. https://docs.qgis.org/3.34/en/docs/training_manual/spatial_databases/simple_feature_model.html, 2024. Accessed on: 2024-05-16.

[18] John Herring et al. Opengis® implementation standard for geographic information-simple feature access-part 1: Common architecture [corrigendum]. 2011.

[19] Konstantinos Pantelios. Development of a QGIS plugin for the CityGML 3D City Database. 2022.

[20] Thomas H. Kolbe, Tatjana Kutzner, Carl Stephen Smyth, Claus Nagel, Carsten Roensdorf, and Charles Heazel. OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard. Open Geospatial Consortium, 2021. http://www.opengis.net/doc/IS/CityGML-1/3.0.

[21] Tatjana Kutzner, Kanishk Chaturvedi, and Thomas H Kolbe. CityGML 3.0: New functions open up new applications. *PFG–Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, 88(1):43–61, 2020.

[22] Claus Nagel, Bruno Willenborg, Zhihang Yao, and Benedikt Schwab. citydb-tool. https://github.com/3dcitydb/citydb-tool/releases, 2024.

[23] John R. Herring. The OpenGIS Abstract Specification, Topic 1: Feature Geometry (ISO 19107 Spatial Schema). volume 5. Open Geospatial Consortium, 2001.

[24] John F. Hughes, Andries Van Dam, Morgan Mcguire, David F. Sklar, James D. Foley, Steven K. Feiner, and Kurt Akeley. *Computer graphics: principles and practice*. Pearson Education, 2014.

[25] Charles Heazel. OGC City Geography Markup Language (CityGML) 3.0 Conceptual Model Users Guide, 2021. https://docs.ogc.org/guides/20-066.html.

[26] Gerhard Gröger and Lutz Plümer. CityGML – Interoperable semantic 3D city models. *ISPRS Journal of Photogrammetry and Remote Sensing*, 71:12–33, 2012. ISSN 0924-2716. doi: https://doi.org/10.1016/j.isprsjprs.2012.04.004. URL https://www.sciencedirect.com/science/article/pii/S0924271612000779.

[27] CodeList. Index of citygml-swg CodeList Examples 3.0.0, 2023. https://data.ogc.org/citygml-swg/CodeList_Examples_3.0.0/. Accessed on: 2024-05-16.

[28] Tatjana Kutzner, Carl Stephen Smyth, Claus Nagel, Volker Coors, Diego Vinasco-Alvarez, Nobuhiro Ishimaru, Zhihang Yao, Charles Heazel, and Thomas H. Kolbe. OGC City Geography Markup Language (CityGML) Part 2: GML Encoding Standard. Open Geospatial Consortium, 2023. http://www.opengis.net/doc/IS/CityGML-2/3.0.

[29] PostgreSQL Global Development Group. PL/pgSQL - SQL Procedural Language Overview. https://www.postgresql.org/docs/current/plpgsql-overview.html, 2024. Accessed on: 2024-05-16.

[30] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book (2nd Edition)*. Pearson Prentice Hall, 2009. ISBN 978-0131873254.

[31] Torben Jastrow and Thomas Preuss. The entity-attribute-value data model in a multi-tenant shared data environment. In *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 494–497. IEEE, 2015.

[32] PostgreSQL-Tutorial. PostgreSQL Views. https://www.postgresqltutorial.com/postgresql-views/, 2024. Accessed on: 2024-05-17.

[33] PostgreSQL-Tutorial. PostgreSQL Materialized Views. https://www.postgresqltutorial.com/postgresql-views/postgresql-materialized-views/, 2024. Accessed on: 2024-05-17.

[34] Panagiotis Peter A Vretanos. OGC® Web Feature Service 2.0 Interface Standard–With Corrigendum, Version 2.0. 2. . 2014.

[35] Esri. What is a shapefile? https://desktop.arcgis.com/en/arcmap/latest/manage-data/shapefiles/what-is-a-shapefile.htm#:~:text=A%20shapefile%20is%20a%20simple,%2C%20or%20polygons%20(areas)., 2021. Accessed on: 2024-07-08.

[36] PostgreSQL Global Development Group. PostgreSQL: Documentation: 16: F.43. tablefunc — functions that return tables (crosstab and others). https://www.postgresql.org/docs/16/tablefunc.html#TABLEFUNC-FUNCTIONS-CROSSTAB-TEXT, 2024. Accessed on: 2024-05-19.

[37] 3D Geoinformation Research Group, Delft University of Technology. Rijssen-Holten Dataset. https://github.com/tudelft3d/Testbed4UBEM, 2024. Accessed: 2023-12-13.

[38] Geospatial Information Authority of Japan. Tokyo 23-ku citygml dataset 2020. https://www.geospatial.jp/ckan/dataset/plateau-tokyo23ku-citygml-2020, 2020. Accessed: 2024-05-06.

[39] C. León-Sánchez, G. Agugiaro, and J. Stoter. Creation of a CityGML-based 3D city model testbed for energy-related applications. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVIII-4/W5-2022:97–103, OCT 2022. ISSN 2194-9034. doi: 10.5194/isprs-archives-xlviii-4-w5-2022-97-2022. URL https://dx.doi.org/10.5194/isprs-archives-xlviii-4-w5-2022-97-2022.

[40] G. Agugiaro. First steps towards an integrated CityGML-based 3D model of Vienna. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, III-4: 139–146, JUN 2016. ISSN 2194-9050. doi: 10.5194/isprsannals-iii-4-139-2016. URL https://dx.doi.org/10.5194/isprsannals-iii-4-139-2016.

[41] City of Vienna. Vienna Dataset. https://www.wien.gv.at/stadtentwicklung/stadtvermessung/geodaten/viewer/geodatendownload.html, 2024. Accessed: 2023-12-13.

[42] Institute for Automation and Applied Computer Science (IAI), Karlsruhe Institute of Technology (KIT). FZK-Haus Dataset. https://www.citygmlwiki.org/index.php?title=FZK_Haus_CityGML_30, 2024. Accessed: 2024-06-26.

[43] Chair of Geoinformatics, Technical University of Munich. TUM CityGML 3.0 Transportation Example Dataset. https://github.com/tum-gis/citygml3.0-transportation-examples, 2024. Accessed: 2024-06-26.

[44] PostGIS Development Team. Postgis st_affine. https://postgis.net/docs/ST_Affine.html, 2024. Accessed on: 2024-06-20.

[45] PostGIS Development Team. Postgis st_translate. https://postgis.net/docs/ST_Translate.html, 2024. Accessed on: 2024-06-20.

[46] Ministry of Land and Tourism (MLIT), Japan. Digital National Land Information GML Data List: Landslide Disaster Warning Area data. https://nlftp.mlit.go.jp/ksj/gml/datalist/KsjTmplt-A33-v2_0.html, 2024. Accessed on: 2024-07-28.

[47] PostgreSQL Global Development Group. PostgreSQL Documentation: CREATE TRIGGER. https://www.postgresql.org/docs/current/sql-createtrigger.html, 2024. Accessed on: 2024-08-07.

[48] Inc. Japan SRA OSS. pg_ivm: An extension of PostgreSQL for Incremental View Maintenance (IVM). https://github.com/sraoss/pg_ivm, 2024. Accessed on: 2024-08-06.

[49] PostgreSQL Global Development Group. pg_IVM 1.9 released. https://www.postgresql.org/about/news/pg_ivm-19-released-2902/, 2024. Accessed on: 2024-09-04.

[50] BRGM and European Union's Earth observation programme Copernicus. Qgis gml application schema toolbox (gmlas). https://brgm.github.io/gml_application_schema_toolbox/, 2021. Accessed on: 2024-09-12.

[51] Even Rouault. QGIS Enhancement: Support for Complex Features in WFS provider. https://github.com/qgis/QGIS-Enhancement-Proposals/issues/277, 2023. Accessed on: 2024-09-12.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.