Expandable Urban Knowledge Graphs: A Use Case in CityGML with OpenStreetMap Data

Scientific work for obtaining the academic degree
Bachelor of Science (B.Sc.)
at the Chair of Geoinformatics
of the TUM School of Engineering and Design
at the Technical University of Munich

Supervised by	M.Sc. Son H. Nguyen Univ.-Prof. Dr. rer.nat. Thomas H. Kolbe Chair of Geoinformatics
Submitted by	Julia Katharina Quarg Bauerstraße 18 80796 München
Submitted on	October 29, 2024

This is a slightly updated version to correct minor grammar and spelling mistakes.

Chair of Geoinformatics
TUM School of Engineering and Design
Technical University of Munich

Abstract

This thesis aims to develop and implement a method for integrating additional data into a semantic city model. Specifically, the graph Database Management System (DBMS) Neo4j is used to expand a City Geography Markup Language (CityGML) data set, thereby creating an expandable urban knowledge graph. CityGML, an international standard for 3D city and landscape models, captures both geometric and semantic information, making it highly useful for urban planning and analysis. However, integrating additional data into CityGML presents challenges due to the complexity of the existing CityGML schema and differences in how data sets represent semantic and spatial features. To address this, a novel approach is proposed using Neo4j, a flexible graph database system, which supports a schemaless structure, allowing the integration of new data and relationships more easily than traditional methods. Additionally, data modelled as a Knowledge Graph (KG) lends itself to easy ad-hoc querying, making it possible to instantly analyse the expanded graph.

The thesis outlines a concept that allows users without deep technical knowledge of CityGML or its schema to integrate data thematically relevant to them and query the enriched graph for their specific use cases. These queries can include both the filtering of thematic attributes like building functions as well as spatial ones, for example distance and area calculations. The concept was implemented by enriching a CityGML data set of downtown Munich with information from OpenStreetMap (OSM) buildings and Points of Interest (POIs). Through the use and development of spatial matching and KG integration algorithms, the CityGML knowledge graph was expanded with additional data. A Graphical User Interface (GUI) was also developed, helping users to thematically and spatially query the expanded graph database and instantly visualize the results without requiring in-depth knowledge of databases or query languages.

The results showed that this approach is effective for data enrichment, enabling both spatial and semantic querying of the extended knowledge graph, as well as intuitive visualisation of the query results. However, limitations remain, such as the uncertainty of applicability in regard to highly specialized data or CityGML data sets from different regions with varying standards. Possible additions and adjustments to address these limitations are also discussed.

Chair of Geoinformatics
TUM School of Engineering and Design
Technical University of Munich

Erklärung

Ich versichere hiermit, dass ich die von mir eingereichte Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

München, den 29. Oktober 2024

Julia Quarg, B.Sc.

Contents

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
ADE	Application Domain Extension
ALKIS	Amtliches Liegenschaftskatasterinformationssystem
Bbox	Bounding Box
CityGML	City Geography Markup Language
CRS	Coordinate Reference System
DAG	Directed Acyclic Graph
DB	database
DBMS	Database Management System
GIS	Geographic Information System
GML	Geography Markup Language
GUI	Graphical User Interface
IQR	Interquartile Range
KG	Knowledge Graph
LOD	Level of Detail
MBR	Minimum Bounding Rectangle
NoSQL	Not only Structured Query Language
OGC	Open Geospatial Consortium
OSM	OpenStreetMap
POFW	Points of Worship
POI	Point of Interest
RAM	Random Access Memory
RDBMS	Relational Database Management System
RDF	Resource Description Framework
SQL	Structured Query Language
STR-tree	Sort Tile Recursive tree
UML	Unified Modeling Language
UUID	Universally Unique Identifier
vCPU	virtual Central Processing Units
VGI	Volunteered Geographical Information
VR	Virtual Reality
XML	Extensible Markup Language

1 Introduction and Motivation

1.1 Introduction

1.1.1 Motivation

Since 2008 CityGML is an international Open Geospatial Consortium (OGC) encoding standard and data model for storing and exchanging 3D city and landscape models. Comprehensive digital representation is achieved through the definition of the most relevant topographic objects, like buildings and roads, in varying Levels of Detail (LODs) [1]. In contrast to pure graphical models, the CityGML data model is not only a standardized way to represent and classify its objects by geometric data, but also focuses on semantics associated with the data [2]. For example, a building in CityGML can have information about its function or its relationships with other objects. Specifically, a building could be stored with attributes providing its address, its function, or its owner to name a few, and with connections describing its relationship to other objects like being located next to a specific road or consisting of multiple building parts. This ability of CityGML to capture both geometric and semantic aspects of city models and to lend itself to high quality visualisation using software like the 3D City Database's (3DCityDB) Web Map Client leads to a wide range of possible applications, from urban planning and environmental simulations to energy estimations and traffic navigation [3].

As the availability of new spatial and semantic data from other sources continues to grow, integrating this additional information into CityGML can significantly expand its applications. For instance, adding details about the location of fire hydrants, which buildings offer public toilet access, or information about local businesses, including their opening hours and contact details, would enhance the possibilities of use for CityGML. An enriched data set could support numerous new use cases that rely on specific semantic information.

Geography Markup Language (GML) is an OGC standard, which is formally specified by an Extensible Markup Language (XML) Schema that predefines the included objects and their relationships. These XML files can be stored in several databases, most prominently 3DCityDB. As a result, the possibilities of adding new data directly to CityGML are limited and require a predefined schema. The two most popular expansion methods are creating generic attributes and objects, and building an entire Application Domain Extension (ADE) [4]. The first method, creating generic attributes and objects, reduces semantic interoperability because there is no way to validate these elements against the CityGML schema [4]. The second method is the development of an ADE by generating an additional XML schema, which requires a good understanding of CityGML and ADEs, since there is, "besides the CityGML standard, no authoritative publication focused on the ADE concept nor a '101 guide'" [4]. In short, these direct ways of adding information require deep knowledge of CityGML, XML and prior knowledge of the additional data.

In comparison, a graph-based DBMS, which stores its content in graph-form, has a very flexible to no set schema, which means it can constantly be adjusted and expanded, but keeping the database semantically intact is left up to the user. CityGML data sets can be effectively stored and queried using a graph-based database because their semantics are defined as a collection of Unified Modeling Language (UML) diagrams

together with the definition of all concepts represented as entities in the UML diagrams and their meanings in the standard document [5], allowing them to be viewed as KGs [6]. According to [7], a KG is "a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities." In the context of CityGML, the nodes could represent instances of buildings, streets, or surfaces, while the edges could represent their relationships, such as a surface belonging to a certain building. Since KGs offer numerous advantages—such as flexibility due to the lack of a fixed schema [8], a concise and intuitive abstraction capable of representing complex relationships [9], and scalable frameworks that allow focusing only on relevant parts of the graph [10]—they have garnered significant attention lately [7]. Consequently, this thesis will use the term KG. Additionally, the idea of using KGs to enrich CityGML data was already used in other papers [6]. All these points make a graph database a good platform to expand the CityGML data set and to consequently run an analysis on the enriched graph.

Given that there is a method for semantically and spatially lossless representation of CityGML data sets in graph form using a graph database [11], namely Neo4j, this paper will build upon these already developed (knowledge) graphs. And since [11] uses the CityGML building module as a working example and these buildings are recognized as "the most detailed thematic concept of CityGML" [2], this thesis will also concentrate on that module. However, the approach is transferable to other thematic modules such as bridges, vegetation, and land use, and the underlying representation method is already being used in that context [12].

This thesis aims to develop a concept, which will allow users without extensive knowledge of the CityGML structure or ADEs to fully integrate their own thematic data into the CityGML graph and consequently query the extended graph, through both thematic and spatial filtering, in the context of their specific use case. To connect the new thematic data to the correct building, spatial matching and therefore spatial data may be required. The concept will be implemented by integrating freely available OSM building and POI data into the CityGML graph of Munich, Germany [13]. Additionally, a simple GUI will be built to enable visualised queries on the expanded database for users without any knowledge about the database and its workings. The goal of this thesis is to develop a standardized framework to expand the knowledge graph of a city for the purpose of thematically and spatially analysing specific use case data.

1.1.2 Problem Statement

A graph database stores its information in graph form, thus splitting the information very intuitively into nodes (for objects/entities), which are normally displayed as circles with a label to categorize the node as well as attributes to store information about the node, and edges (the relationships between the nodes), displayed as arrows. Furthermore, these edges can have a label and direction to describe the exact type of relationship. Therefore, a typical problem in this thesis, namely adding new information to a building by creating new nodes and edges, might be represented in Figure 1.1.

The newly retrieved building information, in this case from OSM, needs to be assigned to the building that it is supposed to enrich with additional information. Two main steps need to be taken in the graph database to achieve this:

1) Turning the new OSM data into nodes or subgraphs (a graph formed from a subset of nodes and edges of the graph), the choice depending amongst others on the amount of added data per building and the intended further usage of the extended graph, in the graph database. 2) Creating an edge to connect the new node or subgraph to the corresponding building node. Depending on user preference, the edge can either be directed from the OSM node/subgraph to the CityGML building node (useful for efficient queries that do not include

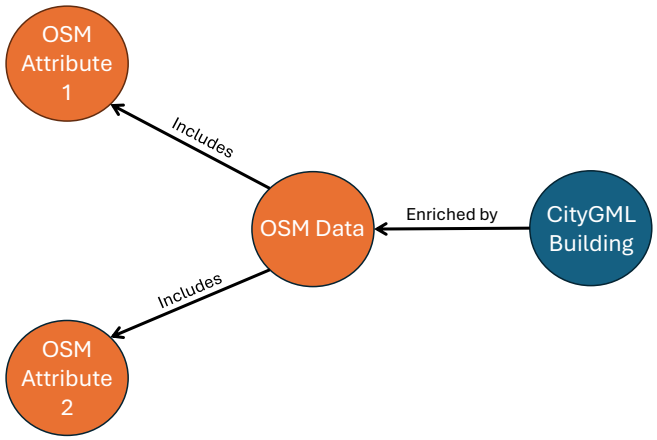


Figure 1.1: Subgraph containing OSM data (orange) connected to a CityGML building (blue).

the new OSM data) or from the building node to the OSM node/subgraph (useful for accessing the new OSM nodes from the direction of the building nodes).

The first point can be expected to be relatively straight forward, since the graph database Neo4j has its own "intuitive query language" [14], providing a good solution for the construction of new nodes. The second part on the other hand is expected to be the core problem for this thesis, since it is not immediately clear, to which CityGML building the new OSM data belongs. To match the new OSM building node with the correct building in the graph, one could try to match the building identifiers, but an "identifier is identical for all versions of the same real-world object" [3] only in the context of CityGML or OSM, not extending to other sets of data. Problematically, OSM and to some extent CityGML both have unstable IDs, which can change when the referenced object is changed [15], making it completely unsuitable for ID matching. Other semantic matches based on address, for example, are also not feasible, since this information is mostly incomplete in both sets of data. The only data consistently available for all buildings is their spatial information in the form of bounding boxes or building surfaces. This leads to a more refined version of the problem shown in Figure 1.2:

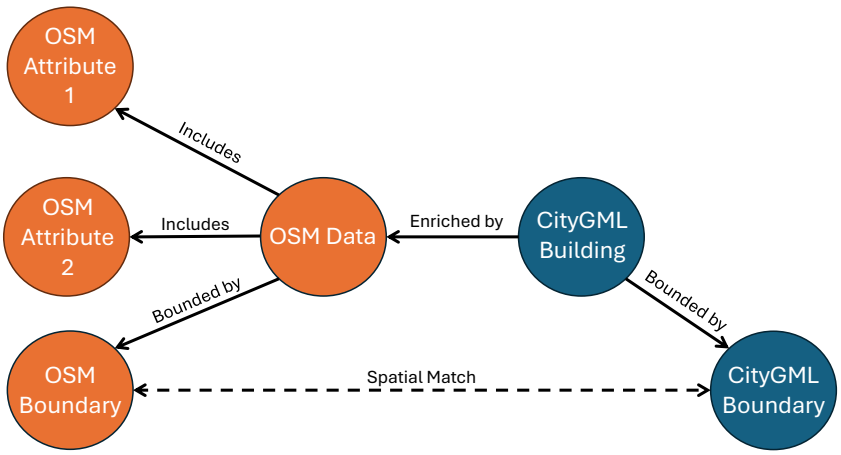


Figure 1.2: Subgraph containing OSM data (orange) connected to a CityGML building (blue) through spatial matching.

The newly retrieved building information will be turned into nodes in the graph database. The spatial data referenced by these nodes as their bounding surface will be compared to the bounding boxes of the CityGML buildings to find the most likely matches. The conclusiveness of matching spatial information will depend on the degree of similarity between the geospatial data of the additional data and of CityGML. In the case of OSM data a visual inspection of a sample area in ArcGIS revealed that while the positions of the CityGML and the OSM geometries on the map match up well, the CityGML geometries are axis-aligned bounding boxes and the OSM geometries are polygons following the outline of the building, making a complete or near complete overlap of both geometries very unlikely. Concepts need to be developed to deal with inconclusive matches, for example multiple OSM buildings matching with one CityGML building and vice versa. The validity of the matches will subsequently be evaluated through various methods, including visualisation in ArcGIS Pro. Finally, the conclusively matched nodes or subgraphs will be connected through an edge describing the new relationship.

1.2 Methodology

1.2.1 Overview

The thesis consists of 3 overall parts.

- Part one contains an introduction to the topic, including motivation, a review of chosen software and relevant literature, and the necessary basics of graph theory; in particular, this section includes a closer look at the advantages and disadvantages of graph DBMS and the resulting implications on what use cases they are suited for.
- Part two is the focus of the thesis and contains the conceptual development of the data integration algorithm. This includes a section on how to retrieve data from a graph database, an extensive section on how to match CityGML and OSM building data using their bounding geometries, while minimizing the risk of false matches. And finally, one last section that focuses on integrating new data into the graph database; additionally, the algorithm is tested via implementation with Python.
- Part three consists of the development of a GUI; the goal here is to show some possibilities of querying the newly integrated data by building a web-based Interface using Python and Cypher.

1.2.2 Literature Review

Since the field of using graph-based approaches to enrich CityGML is well explored using Resource Description Framework (RDF) triple stores [6], there are a few tools and concepts that will be used in this paper, including but not necessarily limited to:

- CityGML, the OGC standard for representing 3D digital city models including building data [2],
- Neo4j, the graph database employed for this project [16],
- Cypher, the declarative graph query language used to query Neo4j, providing expressive and efficient queries for property graphs [14],
- OSM, an open-source project to collect and provide free geodata [17].

Also, a few studies and papers in this field of research need to be considered for this thesis:

- As a basis, this thesis builds on Son Nguyens papers on Spatio-semantic Comparison of 3D City Models [11], which provides the tools [13] to use CityGML data mapped in a graph database. But while that paper and its successors [12] focused on detecting and interpreting changes in the city model, this thesis will focus on enriching that city model by adding data from external sources like OSM.
- Additionally, there are some papers discussing how to enrich CityGML with external data. In their paper “A semantic graph database for the interoperability of 3D GIS data”, [18] the authors use a graph database to fuse the CityGML model with the IndoorGML model for the city of Bologna into one graph to make queries on a semantically connected model containing both CityGML and IndoorGML structures possible. But, since CityGML and IndoorGML are already connected by the Cellspace element in the IndoorGML model directly referencing the ID element of the respective building in CityGML [19], they did not need to solve the problem of fusing or integrating data with heterogeneous spatial information from different sources. The paper does however conclude that for the purpose of enriching CityGML, a graph-based approach is a good option, since it preserves semantic correctness and considerably improves the performance of data management [18]. [20] also worked with IndoorGML, to build an automated process to store IndoorGML in the graph database, thus enabling more diverse scenario-based testing and, more relevant to this thesis, enriching the database by integrating other data sources. Another paper worth considering is an approach to “Integrat[e] 3D City Data through Knowledge Graphs” [6]. While this paper uses a series of different tools like PostgreSQL database (DB) and Ontop to work with RDF Triple Stores instead of a graph database to develop knowledge graphs and integrate 3D city data, it does provide a possible approach on using knowledge graphs to integrate heterogeneous city models, such as OSM and CityGML through comparison of their spatial data. This paper also provides a conceptual approach on how to handle inconclusive matches between OSM and CityGML buildings, but since it uses the CityGML ground surfaces instead of the bounding surfaces, an examination of the extent of its potential applicability is necessary.
- Finally, there are some papers on how to more generally enrich semantic city models. [21] together with [22] discuss how to enrich a 3D City Model with data from Wikipedia, Geonomes and OSM. Location information of the data is isolated in the form of point coordinates, which are then geometrically matched to building polygons in the City Model using fuzzy set theory and so called membership functions [23]. While these papers do not use a graph based approach, their methodology for geometry-matching is worth considering, especially if the enriching data is purely point based. Furthermore, the paper provides an overview over the different possible cases to consider concerning where a POI can be located in reference to a building, and how each of these cases should be handled. [24] focuses on using Volunteered Geographical Information (VGI) like OSM in order to more accurately estimate residential data. With publicly available data like Amtliches Liegenschaftskatasterinformationssystem (ALKIS) only giving a general classification of residential or non-residential for a building, the authors argue that situations like residences above a shop in the same building get overlooked in public data, leading to over- and underestimation of the actual residential space. To reduce false estimations, the paper proposes an approach to integrate OSM data into a 3D building model to identify these cases of split building use. While the paper does not work with CityGML, it gives a great overview over OSM, relevant preprocessing of OSM data and possibilities of its geometric integration using ArcGIS. Complementary to papers discussing data integration into city models, there are also papers geared towards improving/automating that same process. For example [25], who aim to improve upon the process of data integration

by providing a more automated way to collect, integrate and enrich urban open data sets. While this paper uses the concept of RDF Triple Scores for data integration, the actual focus lies on using regression models to predict missing values in the data and not on the integration of new data into a city model. While the topic of filling the holes left by missing values plays an important part in the enrichment of city models, it is a subject outside the scope of this thesis. It should be mentioned however, that the paper calls for development of a more automated mapping process when integrating new data into a city model. Lastly [26], propose an entirely different approach to enriching 3D city models, where instead of integrating the additional data, the user can link it to the correct feature of the city model. While highly practical for certain use cases such as archaeology, where it enables the user to link entire multimedia documents [26], the user has to link the data manually, which is not feasible for enriching city models with thousands or millions of nodes at once. Additionally, since the data is not actually integrated into the model, semantic queries on the model that include the new data are not possible.

1.2.3 Concept Development

The overall concept of this thesis can be split into 4 parts:

1. **Accessing the spatial information of the buildings that are supposed to be enriched by OSM information through spatial matching:** This part includes analysing the CityGML graph in Neo4j to identify the relevant spatial information and its position in the graph, querying the graph for the identified data and storing the queried data in a data format suitable for geospatial analysis.

2. **Identifying and preparing the OSM data:** In this part the OSM data for the chosen region needs to be downloaded, reduced by implementing a bounding box of the relevant area, since the download of OSM data is only available for predefined larger regions, and also stored in a format suitable for spatial analysis. Furthermore, it is necessary to identify the spatial information necessary to match the buildings, and the relevant semantic information to enrich the CityGML graph, because these are stored in separate layers in OSM [27]. Additionally, a coordinate transformation from the initial Coordinate Reference System (CRS), EPSG:4326, to the one of the CityGML data set used in this thesis, EPSG:25832, is necessary.

3. **Matching the OSM buildings to the corresponding CityGML buildings:** As already described in the Subsection 1.1.2, the third part of the concept will aim to develop an algorithm that optimally matches buildings by comparing the building outlines provided by each data set. To that end, all overlapping outlines are identified first by using a spatial indexing algorithm like R-tree, recognizing them as possible matches. Afterwards, an inspection of all the possible matches will be carried out to categorise them into all possible matching scenarios. After identifying each category scenario, a rule needs to be developed on what is considered as an actual match for that category. The goal of this section is an efficient way to spatially match OSM and CityGML data, while keeping the fraction of mismatches low, as well as a list containing all actual matches found.

4. **Integrating the new OSM data into the CityGML graph:** By using Cypher quries, the semantic data for every building taken from OSM will be created as new nodes or subgraphs in the graph database and connected through edges to their matched CityGML building node. This part will also include a concept on how to best store the OSM data in the database and if OSM subgraphs should be developed.

For the implementation of the concept, the Neo4j querying language Cypher and the programming language Python will be used, as well as the Neo4j Driver for Python to remotely access the Neo4j database via its Bolt-protocol. The results will be tested by visualizing them in ArcGIS Pro and manually inspecting a few of the enriched buildings. Additionally, some evaluation methods will be employed to gauge the quality of the matches. The implementation and the concept are at this stage for CityGML 2.0 but are expected to be transferable to CityGML 3.0. The necessary mapping of CityGML 3.0 data into Neo4j has already been developed [12].

Expected Results The expected Results include:

- Enriched building graphs that can be queried for the new data.
- A guiding concept on how to enrich CityGML using a graph database.
- An evaluation of the quality of the matches in the enriched building graphs.

1.2.4 Visualisation using a GUI

The last part of the thesis will focus on querying the expanded database and building a web-based GUI in Python that can be used to visualise queries on the enriched new graph. A typical use case of the GUI should be to find all buildings sharing a sought-after trait in a certain area, like all buildings above a height of 20 meters or all those with a public toilet. This interface is planned with a drop-down menu to control the

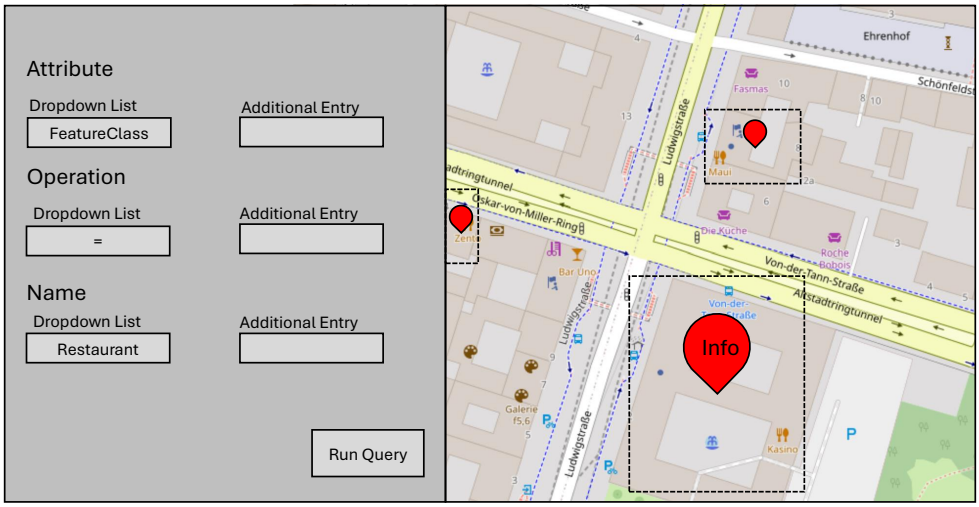


Figure 1.3: Potential realisation of the GUI, map excerpt taken from[17]

possible choices of queries. Additionally, the results of the queries are returned in text form and visualized on a map of the considered area.

Expected Results The expected results include:

- A list of spatial and thematic queries demonstrating the added value of enriching CityGML with information extracted from OSM.

- A successful run of complex queries of both original and enriched data on the enhanced graph to show that semantic interoperability is given within the new graph.
- The developed GUI can be used as a tool for users to help them query enriched CityGML graphs through spatial and semantic filters without any knowledge about the structure of the database.

1.3 Outline

Chapter 1 includes the motivation for this thesis as well as an oversight over its aim and methodology and the literature review. Chapter 2 gives an overview over related concepts, tools and algorithms. The development of the expansion concept can be found in Chapter 3. Chapter 4 describes the setup, results and evaluation of the concept implementation and showcases possible use cases in querying the new graph with and without the GUI. As the final chapter, Chapter 5 consists of the conclusion to this thesis as well as an outlook on possible further developments.

2 Theoretical Background

2.1 Urban Knowledge Graphs

This section introduces the fundamentals of graph theory necessary for this thesis. It then gives a brief overview of some graph applications with a focus on KGs.

2.1.1 Graph Theory

Graph theory, a field within discrete mathematics, focuses on the study of graphs. A graph is composed of a set of vertices, or nodes, representing the objects within the graph, and a set of edges that denote the relationships between these objects by connecting two nodes to each other. A node connected to an edge is called incident to that edge and vice versa, while two edges connected through a node or two nodes connected through an edge are called adjacent. Typically, graphs are visually represented with nodes depicted as dots or circles and edges as connecting lines.

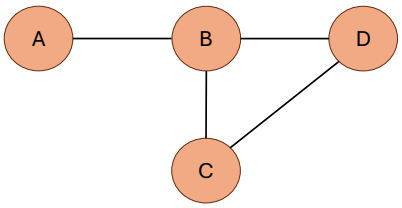
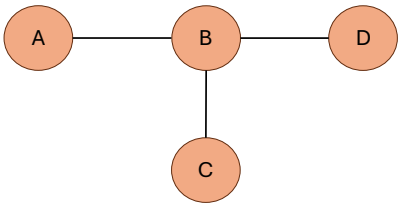


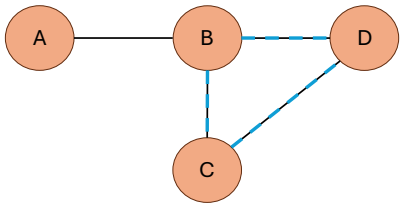
Figure 2.1: Simple undirected graph

The following paragraphs will introduce terms and concepts from the field of graph theory that are relevant for this thesis [28][29]:

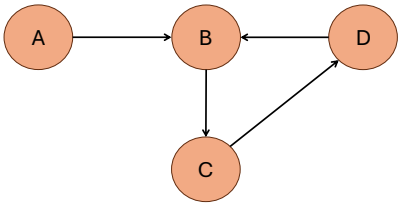
Directed Acyclic Graph (DAG) A **directed graph**, or **digraph**, is a graph in which each edge has a direction, indicating a one-way relationship between two nodes. If no direction is given to the edges, it is called an undirected graph, respectively. Within this type of graphs, a **connected graph** is one in which there is a path between every pair of nodes, ensuring that all nodes are reachable from each other. If every node is reachable from every other node through a directed path, the graph is known as a **strongly connected graph**. An **acyclic graph**, on the other hand, contains no cycles, meaning there is no way to start at a node and follow a sequence of edges that eventually loops back to the starting node. Combining these concepts, a **DAG** is a graph that is both acyclic and directed. This means it has directed edges and no cycles, allowing for a linear ordering of the nodes.



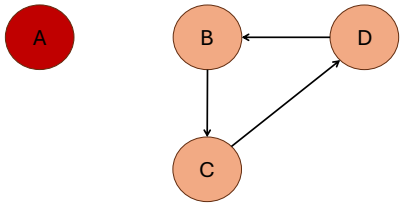
(a) Acyclic graph



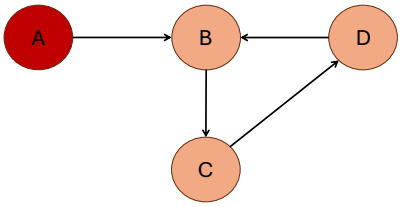
(b) Cyclic Graph



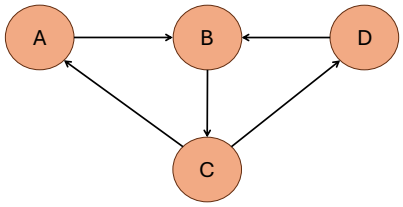
(c) Directed Graph



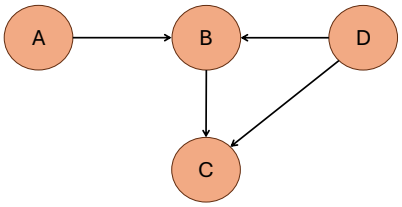
(d) Graph not connected to node A



(e) (Not strongly) connected Graph



(f) Strongly connected Graph



(g) DAG

Figure 2.2: Examples for directed, (a)cyclic, not connected, connected, strongly connected graphs and for DAG

Weights and Labels Labels are identifiers or tags assigned to the nodes (vertices) or edges of a graph. Labels can be used to provide additional information about the nodes or edges, making it easier to distinguish between them or group them thematically. If these labels have numerical, or at least comparable values, they are called weights. The corresponding graph is called a labelled/marked graph or a weighted graph, if the labels are numerical.

Attributes Attributed graphs (or property graphs) are graphs where nodes and edges can have associated attributes or properties. These consist of a property-value pair and provide additional information about the node or edge often in combination with the aforementioned labels and weights [30].

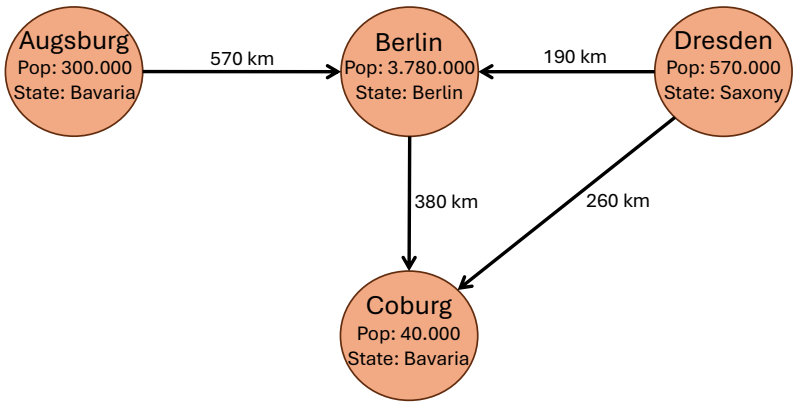


Figure 2.3: Graph displaying data about four German cities: The nodes are labelled with the name of the city and attributed with the city's respective population and state. The edges are weighed with the distances between the connected cities and directed, displaying the possible direction of travel.

In- and Outdegree In a directed graph, indegree is the number of incoming edges to a node, indicating how many edges point to that node. Correspondingly, outdegree is the number of outgoing edges from a node, showing how many edges originate from that node. In an undirected graph the indegree always equals the outdegree and as such is just called the degree of the node. For instance node B of the example graphs has an indegree of 2 and an outdegree of 1 in Figure 2.2f, and a degree of 3 in Figure 2.2b.

Paths A path in a graph is a sequence of edges that connect a sequence of distinct nodes, where each edge is incident to the next node in the sequence. In a directed graph, the path follows the direction of the edges. When looking at Figure 2.2c, one can see that a path from A to D is exists, but no path from D to A.

Subgraphs A subgraph of a graph is formed by taking a subset of nodes and a subset of edges from the original graph, such that the edges in the subgraph only connect nodes also contained within the subgraph. For example, Figure 2.2c shows one possible subgraph for the graph displayed in Figure 2.2f.

Trees Trees are graphs that are undirected, connected and acyclic, meaning there is always exactly one path from one node to another. A spanning tree for a graph is a subgraph of the original graph which contains

all of the graphs vertices while meeting the requirements of a tree. Specifically a minimal spanning tree is the spanning tree of a graph with the smallest possible set of edges. Figure 2.2a shows a minimal spanning tree for all other graphs in Figure 2.2 except for Figure 2.2d.

2.1.2 Graph Applications and [Urban] Knowledge Graphs

A network can generally be regarded as a set of interrelated entities or objects. By understanding the entities as nodes and the interrelations as edges, graphs are used as a mathematical framework to model different types of networks [31]. This graph-based abstraction of knowledge provides multiple advantages, such as intuitive abstractions of complex relationships [9], flexible evolution of data through addition and subtraction of individual nodes and edges [30] and the possibility to use graph databases and graph query languages to analyse aspects of the network, a topic discussed in Section 2.3. What the nodes and edges represent can vary greatly, depending on the network, leading to numerous applications for graph-based data models. A few examples include:

- Social Networks like Facebook or Twitter, in which people are represented as entities and their relationships (Friend, Follower) as edges [32].
- Transit maps, with the nodes representing the stations and the edges representing the connecting train lines [31].
- Logistics networks for communication and electricity, with edges as cables and power lines and nodes representing points of intersection or change [31].

Another use case for graph-based data models that has gained popularity in recent years are KGs [7]. While the term "Knowledge Graph" (or "Semantic Network")is not consistently defined in literature [33][34], the general consensus describes a KG as a graph of real world data with its nodes representing entities of interest and its edges potentially different relations between the entities [7]. The goal of a KG is the structuring of often complex and heterogeneous data from different sources in a meaningful and useful way, often in the context of search engines and machine learning [35]. Prominent examples are the Google KG, which coined the modern term KG in the first place [36] or eBay's Product KG and the aforementioned Facebook social KG among others [37]. Specifically an Urban KG takes urban content such as POIs, buildings and regions as entities (nodes), while semantic and spatial dependencies are modeled as relations (edges) [38].

2.2 CityGML and Semantic City Models

2.2.1 Semantic City Models

"Semantic 3D city models are virtual models of the urban environment, that is, data sets representing the entities of the physical reality like buildings, streets, trees, bridges, and the terrain. In contrast to Virtual Reality (VR) models, they are structured (e.g. subdivided and attributed) according to thematic and logical criteria and not according to graphical or rendering considerations." [3] These models prioritize thematic and logical structuring over graphical considerations, enhancing their utility for analysis and urban planning. The advantages of semantic city models include their ability to integrate various types of data, facilitate interoperability between different systems, and provide a detailed and accurate representation of urban environments that supports informed decision-making, making them particularly useful for tasks such as urban development, environmental simulation, disaster management, and pedestrian navigation [1]. However, their complexity and the need for detailed, high-quality data can make them resource-intensive to create and maintain, which limits their use in projects with constrained budgets or less emphasis on detailed data

integration like Google Earth [3]. Despite these challenges, their structured nature makes them invaluable for comprehensive urban analysis and planning, offering insights that are not possible with purely graphical models.

2.2.2 CityGML

CityGML is an XML-based standard and data model designed for representing and exchanging 3D city models. Originally developed by the Special Interest Group in 2002, CityGML was established by the OGC in 2008 as a standard providing guidelines for how urban data should be structured and encoded [2]. As a data model, CityGML defines the schema for storing not only (3D) geometric and graphical, but also semantic and topological properties of urban features, making it a crucial tool for interoperable data exchange of (semantic) 3D city models [1]. Moreover, CityGML data sets are organized into 5 different LODs (see Figure 2.4), which range from LOD0 (basic 2D footprints) to LOD4 (highly detailed interior models). This hierarchical structuring allows for varying degrees of detail to be used depending on the application and needs of the user.

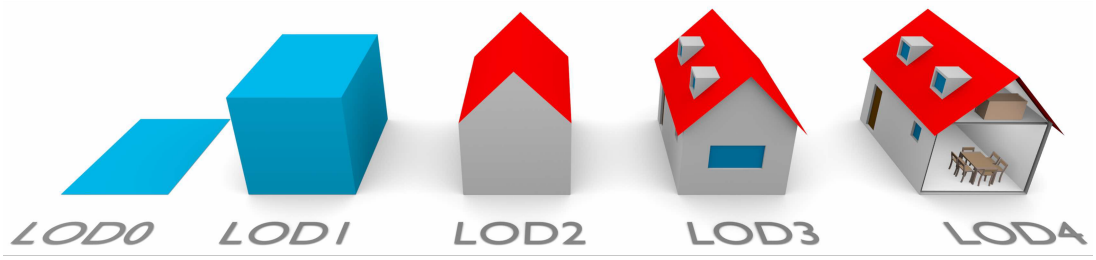


Figure 2.4: Building Representation in 5 Levels of Detail LODs. Source: Copyright ©Delft University of Technology [39]

CityGML data can be directly stored in various formats, including file-based storage (e.g. GML files) and databases like 3DCityDB. It can be visualized using Geographic Information System (GIS) software, 3D modeling tools, or specialized viewers designed for CityGML, most notably the 3DCityDB Web Client [40]. Since the focus of CityGML is on the semantic aspects of 3D city models [1], it is modular, meaning it is divided into different thematic modules that correspond to different aspects of the urban environment. These modules include the core module as well as the extension modules Appearance, Bridge, Building, CityFurniture, CityObjectGroup, Generics, LandUse, Relief, Transportation, Tunnel, Vegetation, WaterBody and TexturedSurface, which are all connected to the core module [2]. As already mentioned in Chapter 1, this thesis will focus on the building module, but is in theory applicable to all other modules. The schema of every module is defined using UML, providing a clear and structured way to represent the relationships and properties of the different urban features. Due to its complex structure with multi-level deep associations (as can be seen in Figure 2.5) and its richness in semantic real world information, CityGML data, and in this case specifically data from the building module, can be regarded as having a KG structure and consequently be mapped into graphs, as was done by [11][6].

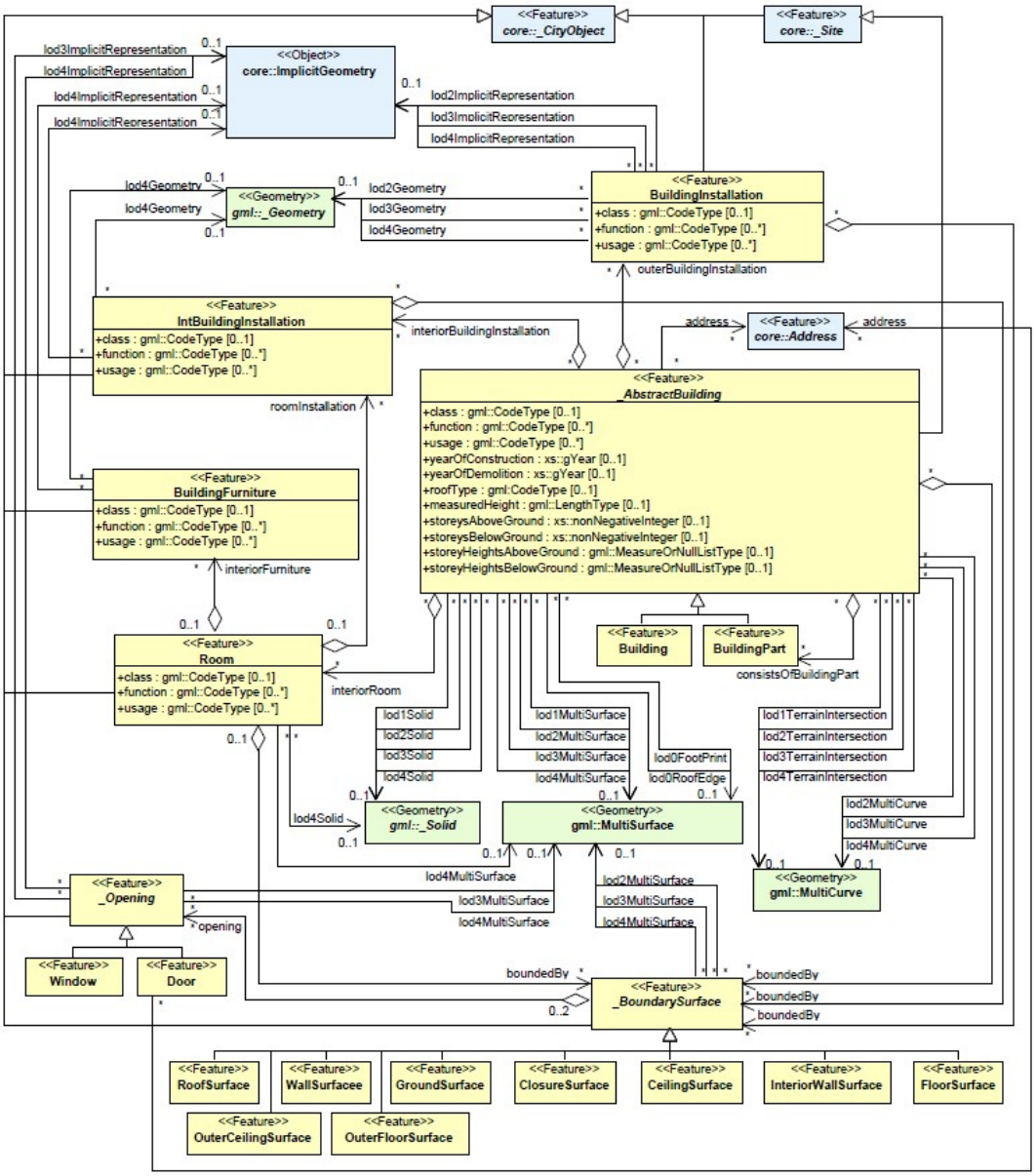


Figure 2.5: UML diagram for the CityGML building module. Source: Copyright ©2012 Open Geospatial Consortium, Inc. [2]

2.3 Graph Databases

The beginning of this section introduces the concept of graph databases, comparing them to relational ones. After that the graph database Neo4j, which is employed in this thesis, is introduced, including an overview over accessing and handling transactions in a Neo4j database.

2.3.1 Relational and Graph Database

Relational Databases

Since its first introduction in 1969, the relational model for databases is the most widely used and popular data model in the world [41]. A database employing the relational model is called a relational database and

the maintaining system is called a Relational Database Management System (RDBMS). Most RDBMS use Structured Query Language (SQL) to access their data. In the relational model, information is stored in tables, or "relations". Each table represents a collection of related data entries, and each row (or "tuple") within a table corresponds to a single record. Columns in a table represent the attributes of the data, and each column holds data of a specific type. For a sustainable relational database the overall model structure as well as the number and the attribute types of columns need to be defined in a preset schema.

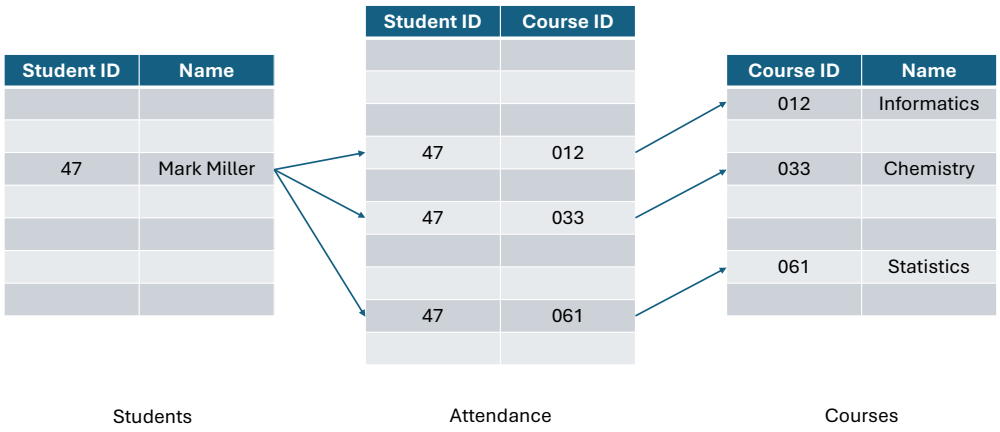


Figure 2.6: Indirect Relationships in a relational database through a JOIN table [adapted from Neo4j Docs [42]]

Relationships between data entries in a relational database are established indirectly through the use of primary keys and foreign keys, requiring them to be non-empty. A primary key is a unique identifier for each record in a table, ensuring that each entry can be distinctly accessed. A foreign key is a column (or set of columns) in one table that refers to the primary key in another table. In case of a many-to-many relationship of the keys, an additional junction (JOIN) table with all referenced foreign keys is required. Consequently, a STUDENT in Figure 2.6 has the primary key Student ID and a course is identified by its Course ID. But since a student can enroll in multiple courses and a course can have multiple students, the foreign keys that connect students and courses are stored in a separate JOIN table. However, creating these junction tables costs memory and computing power and significantly increases query response time, with the delay growing exponentially relative to the table size [11][43]. Furthermore, with a rising depth of relationships between entities, queries with the JOIN operator become more extensive and unintuitive (see Listing 2.3).

Graph Databases

Graph databases, as part of the so-called Not only Structured Query Language (NoSQL) databases, which also include - among others - the RDF triple stores mentioned in Subsection 1.2.2, have been gaining in popularity, particularly in applications that require efficient access to complex relationships within large data sets. They are widely used in areas where relationships are as important, or even more important, than the individual data points, making them particularly popular in scenarios where accessing only a part of the database and its immediate relationships is crucial. This capability makes graph databases highly efficient for traversing and querying connected data without the overhead of complex JOIN operations typical in relational databases [11]. In graph databases, information is stored as a graph using nodes and edges. Each node and edge can have properties, which are key-value pairs that store information about them. As

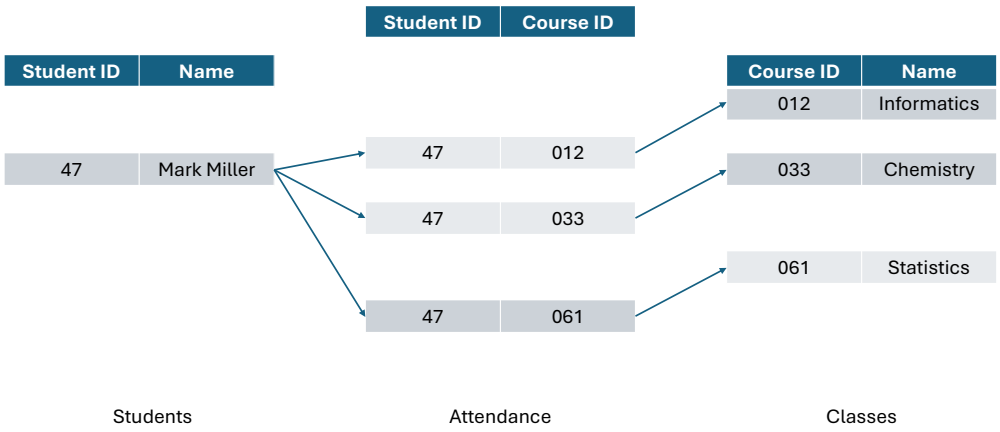


Figure 2.7: Data connected without the need for JOIN operations in a Graph DB setting [adapted from Neo4j Docs [42]]

previously mentioned in Section 2.1, this structure allows for a more natural representation of complex data relationships and makes querying these relationships more straightforward. One of the key features of graph databases is that they place a strong emphasis on relationships, which are direct and explicit. While they do store individual data points (nodes), the primary focus is on how these nodes are interconnected. Edges directly connect nodes, representing relationships without the need for foreign keys or join tables. This direct connection makes traversing the graph to find related nodes extremely fast and efficient, as the database engine can follow edges directly without looking up additional information. Another important feature of graph databases is that, unlike relational databases, they do not require a predetermined schema. This schema-less nature allows for greater flexibility and adaptability, as new types of nodes and edges can be added without altering the existing structure, proving particularly advantageous in dynamic environments where the data model may evolve over time [44].

In conclusion, it becomes clear that neither the relational database nor the graph database can be regarded as superior to the other, but instead that they are suited for different use cases. While relational databases are ideal for huge, flat, homogeneous data structures with small relationship-depth, graph databases are better suited for complex, heterogeneous data structures with numerous and complex relationships.

In the context of this thesis, where the goal is to expand a KG with new data, a graph database provides the ideal setting, since

- Only access to the parts of the CityGML KG which are relevant for the expansion is required.
- The original structure of the KG should not be altered through the expansion of the KG.
- The amount of new data and what relationships will be formed is not clear in advance.

2.3.2 Neo4j

For the process of expanding the CityGML KG, the Neo4j graph database is used. Neo4j is a graph DBMS, which was developed by Neo4j, Inc. and is considered the most popular graph DBMS worldwide [45]. It is a high performance, native graph database, meaning the connections (edges) between nodes are stored as part of the database and don't need to first be computed when queried. This enables high query

speeds because data not relevant to the query can be ignored. Neo4j is also a fully Atomicity, Consistency, Isolation, Durability (ACID) transactional database. Neo4j comes in two on premise versions, Community and Enterprise, and three in-cloud versions, namely AuraDB Free, Professional and Enterprise [16]. For this thesis, the free Community version is employed. The use of Neo4j is additionally advantageous for this thesis in particular, because the CityGML KG was also mapped using Neo4j [13]. Working with Neo4j guarantees that there are no questions of compatibility or integration that could arise by using another graph DBMS, as those may store graphs differently.

Transaction Optimization

ACID-compliant transactions All operations in a (graph) database can be divided into two distinct groups:

- Read operations, where information is retrieved from the graph database, without any changes to the queried graph.
- Write operations, where modifications to the graph in the form of addition, deletion or alterations are implemented.

The fact that Neo4j is a transactional database means that all write operations within the graph database must be performed in a transaction. A transaction can be understood as a thread confined unit of work that is either committed entirely or completely rolled back. All transactions in Neo4j are ACID transactions, guaranteeing [46]:

- Atomicity - If any part of the transaction fails, the database state is left unchanged
- Consistency - A transaction will always leave the database in a consistent state
- Isolation - During a transaction, the data being modified can’t be accessed by other operations
- Durability - the results of a committed transaction can always be recovered by the DBMS

But how does Neo4j guarantee ACID-compliance in its transactions? After the start of a transaction, all affected objects are locked and held in a lock in main memory until the transaction is either marked as successful or failed. In case of success, changes are committed to the database and in case of failure, the entire transaction is rolled back and the database remains unchanged (see Figure 2.8). If another transaction tries to access locked data, it stays pending for resources and if a risk of so called "deadlocks" is detected, an exception is thrown and the new transaction is rolled back. In the case of read operations, Neo4j initiates a read transaction, which allows for consistent reads without locking [46].

Optimizing transactions through batching It is normally best practice to start a new transaction for every node or relationship that is modified, enabling the computer to run as many concurrent transactions as there are virtual Central Processing Units (vCPU), also called threads. This practice avoids long-lasting locks on data and thus on main memory, gives more precise control over error handling and debugging and a clear overview over the application logic. However, every single transition comes with an overhead responsible for the frame of the transaction, including the initializing and committing (or rolling back) of the transaction. These overheads significantly slow down operations on sets with millions of nodes and relationships [47]. In the case of large sets of nodes, there is the option of transaction batching. As the name implies, batch transactions refer to the practice of grouping multiple operations of the same kind into batches, where all operations in a batch are initialized and subsequently committed together. This practice significantly reduces overhead and improves computation time, in some cases by over 90% [47]. However, there are a few things to consider; If one operation in the batch transaction creates a rollback, the entire batch gets rolled back and since all nodes affected by the transaction are locked and held in the computer’s Random Access Memory

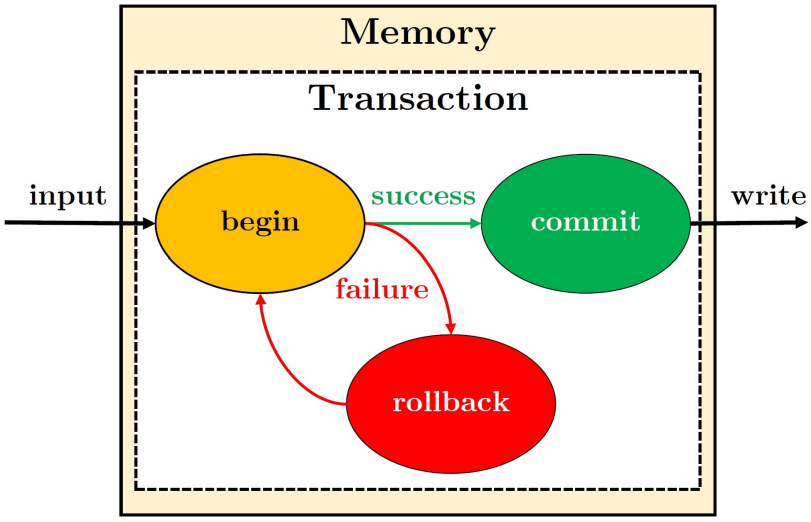


Figure 2.8: Managing transactions in Neo4j by only committing successful transactions to main memory. [Source: [11]]

(RAM), batch transactions are only feasible with enough main memory available [48]. And lastly, the risk of a possible deadlock rises with the longer lock duration. In conclusion, batch transactions can offer significant performance advantages, especially for bulk data operations, but need large amounts of RAM and careful preparations to minimize the risk of deadlocks and subsequent rollbacks [46].

Cypher and the Bolt Protocol

Cypher The Neo4j database provides its own query language, a NoSQL called Cypher [14]. It is a declarative graph query language and allows for efficient and expressive queries. Cypher is visual in its representation, making it intuitive to use.

Listing 2.1: Example Cypher code

```
1 // Nodes are in circular brackets, relationships in square
      brackets, arrows denote directions of edges
2 MATCH (c:Customer)-[p:PLACES]->(o:Order)  // MATCH-clause as a
      visual JOIN-Operator
3 WHERE o.priority = "high"
4 RETURN c.name   // Instead of SELECT
```

Listing 2.2: Comparison of the MERGE and the CREATE clause

```
1 CREATE (c:Customer {name: 'Tom', age: 30})
2 RETURN c;
3 // CREATE always creates a new customer named Tom.
4
5 MERGE (c:Customer {name: 'Tom', age: 30})
6 RETURN c;
7 // MERGE looks to see if a Tom with the given attributes already
      exists, returns the node if he does and creates a new
      customer named Tom in case he does not.
```

It is similar to SQL in many ways, for example using the WHERE statement in the same way, but there are some key differences. Since graphs do not have a schema, Cypher, in contrast to SQL, also does not require

a fixed schema to add new relationships or attributes to objects in the graph. Additionally, Cypher queries are more concise than SQL queries, since they do not use the JOIN operator.

Listing 2.3: Comparison Cypher and SQL

```
// SQL query with 3 JOIN tables
SELECT c.customer_name, p.product_name, p.product_price, oi.
    quantity
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
ORDER BY c.customer_name, o.order_date;

// Cypher query with one MATCH clause
MATCH (c:Customer)-[:PLACED]->(o:Order)-[:CONTAINS]->(oi:
    OrderItem)-[:IS]->(p:Product)
RETURN c.customer_name, p.product_name, p.product_price, oi.
    quantity
ORDER BY c.customer_name, o.order_date
```

All Cypher queries trigger either a read or a write transaction in the database. While a typical read query can be seen in Listing 2.3, Listing 2.2 shows the two ways to construct a write query in Cypher by using either the MERGE or the CREATE clause [49].

Cypher queries can be executed directly from Neo4j's browser-based client or from one of the official Neo4j drivers, including the Neo4j Python Driver, through the use of the Bolt protocol. The Bolt protocol accepts Cypher queries, executes them and returns the results to the driver for further usage in the associated programming language. How the query in Listing 2.1 in Neo4j Python Driver could be executed, can be seen in the example code below.

Listing 2.4: Accessing the Neo4j database through the Neo4j Python Driver

```
// Code will return a list of names of all high priority
    customers.
// Names are returned as records bundled into a result object.
uri = "bolt://localhost:7687"
(username, password) = "neo4j", "password"

driver = GraphDatabase.driver(uri, auth=(username, password))

with driver.session() as session:
    query = """
    MATCH (c:Customer)-[:PLACES]->(o:Order)
    WHERE o.priority = "high"
    RETURN c.name
    """
    result = session.run(query)
    for record in result:
        print(record["c.name"])
driver.close()

```

2.4 OpenStreetMap

OSM is a free and open geographic database maintained by a large, active volunteer community. It is constantly growing, and all its data is provided under a free license. The data quality is described as "quite developed and mature as compared to geodata from commercial vendors" [50] and organisations like Wikipedia and Foursquare use their data commercially.

OSM data is stored as nodes, which are points defined by longitude and latitude coordinates in the EPSG:4326 coordinate reference system, more commonly known as WGS84. Individual nodes represent point features such as POIs, while the more complex line and polygon features are represented as lists of nodes. All features are stored in one coherent database and then sorted into layers upon extraction according to their type. These layers include the building layer, POI layer, Points of Worship (POFW) layer, roads layer, waterways layer, and more. It is important to note that every object in OSM has an ID generated at creation. However, these IDs are not stable and can change if the feature is modified. Additionally, an ID is only unique within the layer to which the object belongs [27].

Direct downloads from OSM are possible, but limited to 50,000 nodes. For larger data sets, OSM provides several endorsed sources, including the Overpass API, Planet OSM, and GEOFABRIK. GEOFABRIK, a consulting and software development firm based in Karlsruhe, offers daily updates and excerpts under the free OSM license. Data for selected regions is available in OSM's own PBF format and as shapefiles [51]. For this thesis, data for Oberbayern, including Munich, will be downloaded from GEOFABRIK as a shapefile.

2.5 R-Tree Algorithms

2.5.1 Standard R-tree

As a dynamic index structure, R-Trees, or Rectangle-trees, are trees designed to support efficient spatial querying and updating, thus making them ideal for applications involving large data sets with spatial attributes, such as points and polygons. Since their introduction in 1948 [52], they have found extensive use in multiple of fields including geographic information systems, computer-aided design and data mining and warehousing [53]. The basic concept of an R-tree [52] is to group spatial objects located near one another together into a larger object internal to the R-tree which contains all their outlines within its aggregated Minimum Bounding Rectangle (MBR). This effectively means that a geometry not intersecting with that MBR will also not intersect with any objects contained within it. Real world objects are represented as leaf nodes in a tree (represented in red in Figure 2.9), with the larger internal object being located one level higher in the tree. All objects on the higher tree level are then aggregated into even bigger objects on the next tree level and so on, until the tree is recursively constructed (see Figure 2.9). The highest level only contains one node, the so called root node. This effectively means that during any queries conduced with the R-tree, such as an intersection query, most irrelevant objects can be avoided, thus making the R-tree very efficient in the context of some spatial queries, such as intersection and nearest neighbor search. Moreover, the average time complexity of search operations in an R-tree is

$$O(\log_M n) \tag{2.1}$$

with M representing the maximum number of objects contained within an internal object and n representing the total number of nodes [11]. In Figure 2.9, this would mean that M and n equal 2 and 11, respectively. Additionally, standard R-trees can be modified during use through insertion or deletion of geometries by recursively splitting parts of the R-tree [52], a property of R-trees not further discussed in this thesis.

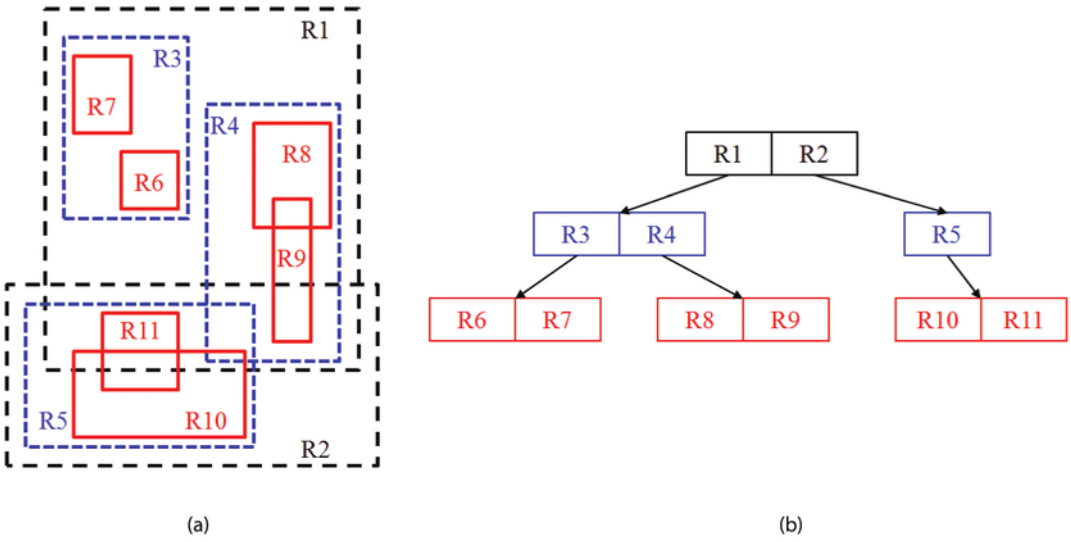


Figure 2.9: R-Tree indexing example: 2D visualization (a), hierarchical dependencies (b). [Source: [54]]

2.5.2 STR-tree Packing Algorithm

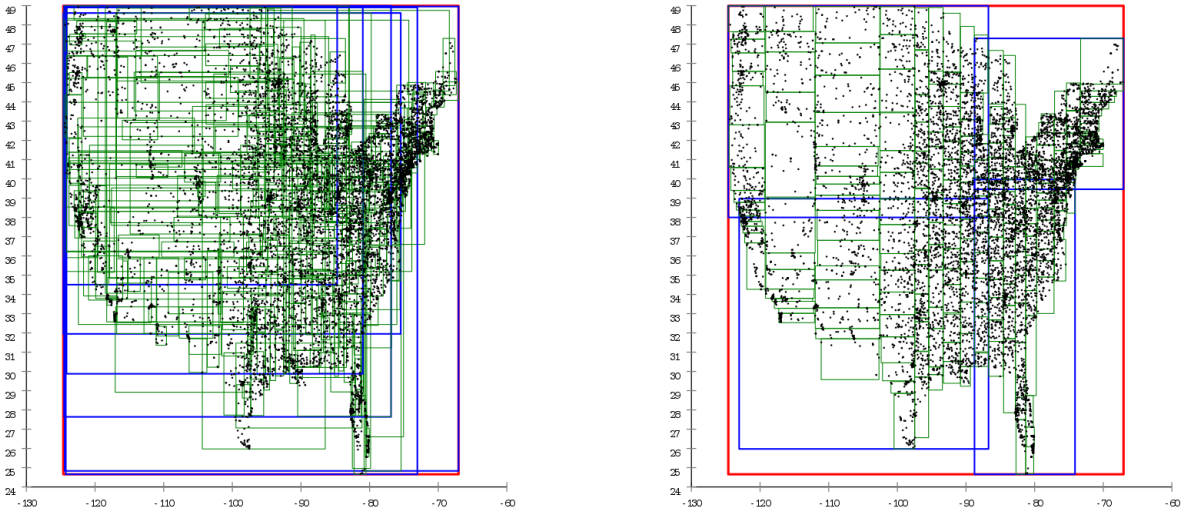
Since the R-tree is widely used, there have been many attempts to optimize it for various use cases. One big point of optimization is how to decide, which neighboring objects get grouped together into one MBR, a process referred to as the packing algorithm of the R-tree. The original packing algorithms for inserting objects into the R-tree, that were suggested by Guttman [52], have several disadvantages, such as "high load time, sub-optimal space utilisation and poor R-tree structure requiring the retrieval of an unduly number of nodes to satisfy a query" [55]. This has led to the introduction of multiple packing algorithms for R-trees, such as R*-tree and STR-tree, which is the packing algorithm employed in this thesis. Through recursively pre-sorting the R-tree into slices, as can be seen in Listing 2.5, the STR-tree minimizes the disadvantages named above. The difference in space utilisation can clearly be observed by looking at Figure 2.10. However, it must be noted that as a static R-tree [53] the STR-tree is unable to split nodes "in balance" [56], meaning that modification during use is not possible.

Listing 2.5: Algorithm for constructing a STR-tree

```
Input: R as a set of rectangular two-dimensional data elements
       N as Number of Elements in R
       M as Maximum number of  child nodes/rectangles per node
Output: R-tree structure

While R > 1:
    # Step 1:
    Calculate minimum number of leaf nodes P = ceil(N/M)
    # Step 2:
    Sort all N rectangles by X coordinate of their center point
    # Step 3:
    Divide sorted dataset into S = ceil(sqrt(P)) vertical slices,
     each with S leaf nodes and S * M rectangles
    Exception: The last slice may contain fewer than S * M
   rectangles
    For each slice:
        # Step 4:
        Sort Rectangles by Y coordinate of their center point
        # Step 5:
        Group every M rectangles into one (leaf) node in sequence
```

```
19          Exception: The last node may contain fewer than M
        rectangles
20       # Step 6:
21       Group the minimum bounding rectangle (MBR) of each (leaf)
        node as a new set of rectangles R
22
23  Return: Root node of the constructed R-tree
```



(a) R-tree with standard Guttman Split

(b) R-tree with STR packing algorithm

Figure 2.10: Standard R-tree on the left versus STR-tree on the right, slices visible [Source: [57][58]]

3 Concept Development

3.1 Procedural Overview

Before getting into the details of the concept developed in this thesis, this section gives a brief overview over the rough logical steps the concept can be divided into. As is visible in Figure 3.1, the concept can be divided into six sequential thematic blocks. In the following, a short summary of each of these steps will be given.

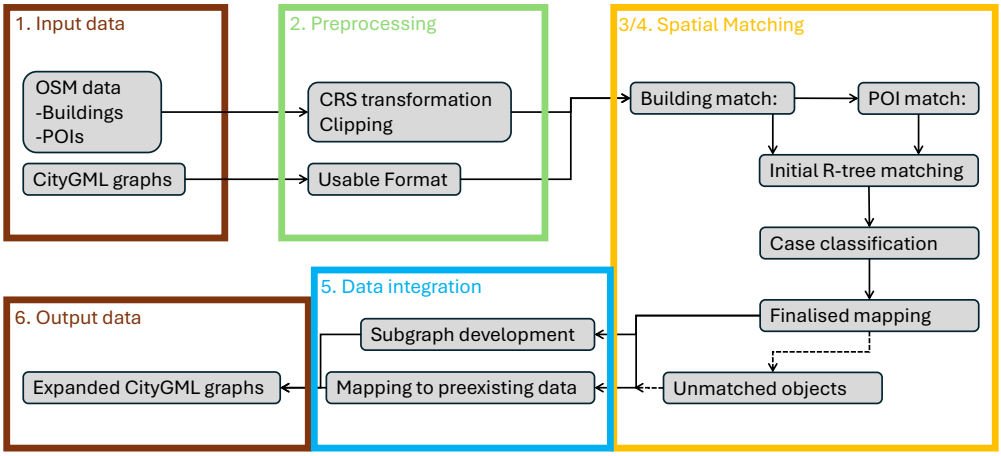


Figure 3.1: General overview over the concept for the integration of OSM data into the CityGML KG

In a first step, the input data from both data sets has to be extracted from the respective sources. This includes identification of relevant input data, both in the context of the CityGML graph and the enriching OSM information, where that input data can be found and how it can be extracted.

Preprocessing is conducted as a second step to ensure the data is converted into a format suitable for further analysis and integration. Here the main priority is to ensure that both sets of information are compatible with one another and the necessary analysis tools. This section includes, but is not limited to, setting both data sets to the same CRS, the same area and the same data format.

Next, the third step in the process is the matching of polygonal geometries, specifically OSM and CityGML buildings through their respective footprints. Due to the fact that there are no universal identifiers for these buildings, a topic discussed in Subsection 1.1.2, spatial matching is necessary. The goal of this section is to match as many OSM footprints to CityGML footprints, while still maintaining a conservative approach. This means minimizing the risk of false matches is prioritized over maximizing the amount of matched buildings. The section is also split into multiple parts and together with part four represents the main focus of this concept. Additionally, it is important to mention, that since the OSM building Polygons do not carry much

semantic information, and some don't carry any (see Table 3.1), their matching can mostly be considered as a preparative step for the matching of OSM POIs and CityGML buildings.

The fourth step focuses on spatially matching point geometries to polygonal ones, in particular OSM POIs to CityGML buildings. These matches are done with the help of an intermediate matching step. First the POI coordinates are matched to OSM building footprints and then, using the already existing matches from step three, the OSM POIs are matched to the CityGML buildings. This additional matching is necessary, due to the different natures of OSM and CityGML footprints (see Subsection 3.3.2). Otherwise, the processes in step four are fairly similar to those in step three.

After the spatial matching is completed, in a fifth step the new data needs to be integrated into the GraphDB. Two main aspects need to be considered during this process: For one, in what form the new data should be added, namely as nodes or subgraphs and how these nodes or subgraphs should be structured. Additionally, correct mapping of the new data nodes to their respectively matched CityGML building node also needs to be ensured.

Finally, in the sixth step, the output data is an enriched CityGML KG with new information expanding the contained buildings. This expanded KG can then be queried for the new data provided by OSM.

This chapter will now consequently deal with all of these steps in order.

3.2 Data Collection and Preprocessing

3.2.1 CityGML Data

Extraction

The goal of this initial step is to identify all CityGML data necessary for spatial matching and subsequent integration, and consequently extract that same data through a Cypher query. For this purpose it is important to examine the structure of the graph database. As previously mentioned, the database comprises nodes and edges that represent various buildings and building attributes and their interrelationships. For instance, a

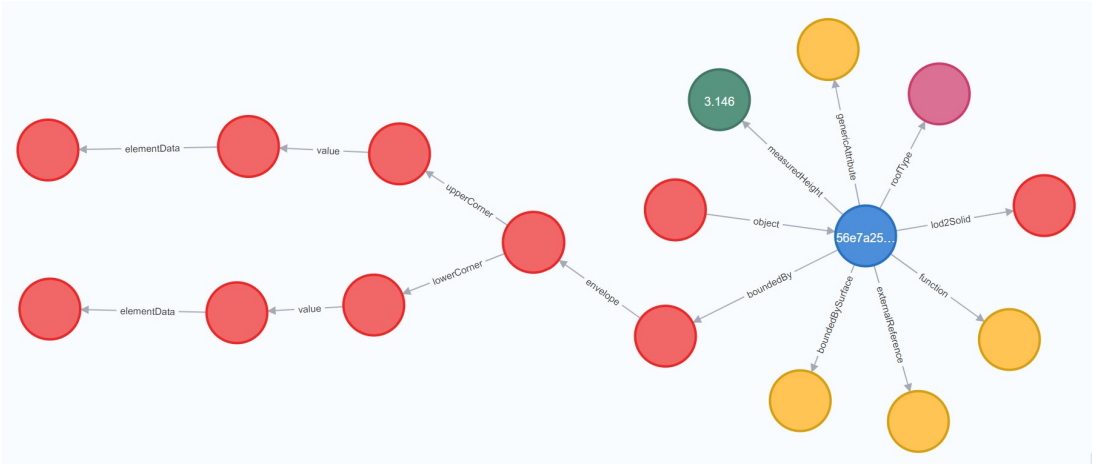


Figure 3.2: Excerpt of a graph from the CityGML Graph DB: The building node is blue and the red nodes containing the Bounding Box (Bbox) coordinates are on the far left of the graph excerpt. All other nodes connected to the buildings central node represent different building attributes, such as height (green), roof type (purple) or building function (yellow).

node may contain the value 3.146, while the edge connecting this node to the corresponding building node might denote that this is its measured height, as illustrated in Figure 3.2. Each building and its attributes

can be understood as a distinct subgraph encapsulating all information specific to that building. All these subgraphs are interconnected through what are referred to as city model nodes[1].

The next step is to identify the data required for the process of expanding the CityGML building graphs. This includes:

- **Building footprint:** A building footprint or boundary to spatially match to the OSM data.
- **Unique Identifier:** A unique building identifier is a combination of numbers and/or letters that allows the user to link back to the original building subgraph in order to accurately map the matched OSM data later on. This identifier can be any attribute of the building, as long as it is unique to that building in the context of the entire data set.

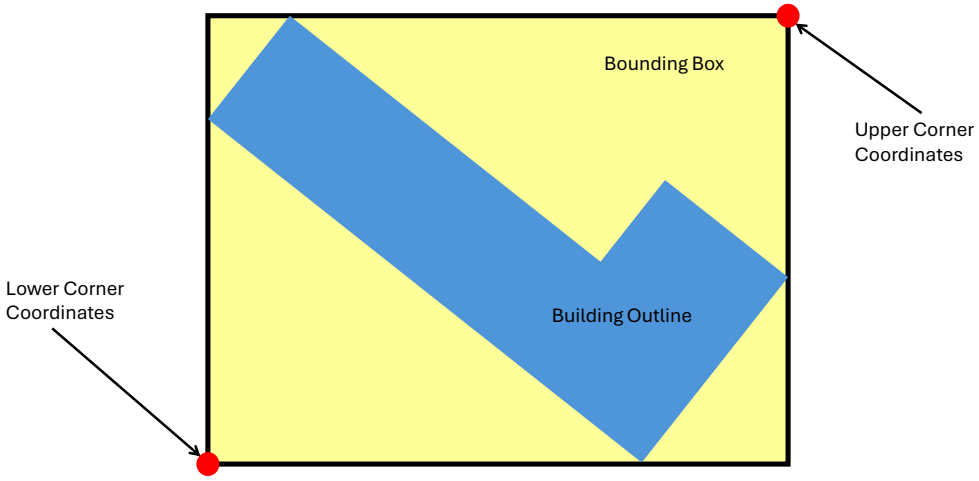


Figure 3.3: Make-up of the Bbox for a CityGML building: The Bbox is the smallest possible axis-aligned rectangle that completely encloses the given building outline, ensuring that all points of the outline are within or on the boundary of the rectangle. It is also known as an MBR. The details of the Bbox can then be stored in just two coordinate pairs, traditionally the lower left and upper right corner of the rectangle. This technique has the significant advantage of requiring minimal storage space, as it only needs two coordinate pairs to represent the spatial extent of the building. Depending on the outline and orientation of the building, the size of the Bbox can reach from the size of the building it encloses to multiple times its size. Generally, it can be stated that buildings with less compact footprints have a bigger size gap between a building and its Bbox.

To represent the footprint, the Bboxes of the buildings are used, stored as four individual coordinate values—two for the lower corner and two for the upper corner (see Figure 3.3)—in two separate nodes (see Figure 3.2). As a consequence of this storage format, the coordinates are extracted as four separate record entries in the return result of the query and not as one cohesive geometry.

Listing 3.1: Cypher code for the extraction of the Bbox data

```
1 MATCH (b.Building)-[:boundedBy]-()-[:envelope]-()- [:lowerCorner
    ]-(
2 )-[:value]-()-[:elementData]-(d.__ARRAY__)
3 RETURN d.`ARRAY_MEMBER[0]`, d.`ARRAY_MEMBER[1]`, b.UUID
4 UNION
```

[1] For additional details on the makeup of the Graph DB, refer to [11] and [13].

```
MATCH (b.Building)-[:boundedBy]-()-[:envelope]-()- [:upperCorner
    ]-(
)-[:value]-()-[:elementData]-(__ARRAY__)
RETURN d.`ARRAY_MEMBER[0]`,d.`ARRAY_MEMBER[1]`, b.UUID
// Array_member[0] refers to the x-coordinate or easting of a
    coordinate pair.
// Array_member[1] refers to the y-coordinate or northing of a
    coordinate pair.
```

For identification purposes, there are two options: utilizing the Universally Unique Identifier (UUID) stored in each building node or relying on the building's geometric data. Consequently, the buildings unique Bbox can provide adequate identification and since they are already being extracted for each building, no additional data extraction would be necessary. However, as previously mentioned, the coordinates are currently represented as four separate values instead of a complete Bbox, and the extraction of an intermediate unique identifier was chosen as an aid for constructing the final Bbox (see Subsubsection 3.2.1). For this purpose, the UUIDs for all buildings are extracted as well. There are more complex methods like a longer MATCH chain that could bypass the auxiliary UUID extraction, but for this thesis, the simplest solution was chosen. A possible Cypher code for extracting all Bbox coordinates and UUIDs can be found in Listing 3.1.

Preprocessing

Before the extracted CityGML data can be used for spatial matching, two preparative steps have to be taken.

As already mentioned in Subsubsection 3.2.1, the Cypher query (see Listing 3.1) does not return a Bbox, but instead four separate values. More specifically, if we consider that the database contains *n* building nodes, the query result would comprise *2n* records, with each record including an UUID, an ARRAY_MEMBER[0] and an ARRAY_MEMBER[1] value. But in order to be able to match building footprints, a rectangle needs to be constructed. A possible method can be seen in Listing 3.2. Since Cypher works strictly sequentially, executing the query line by line, all the records containing coordinates from the lower corner are in the upper, and all records containing coordinates from the lower corner are in the lower half of the results list. Consequentially, through a split after *n* records and subsequent union of both list on the UUID, the four coordinates are now ordered and can be used to create a list of *n* rectangles. Lastly, these rectangles need to be stored. In this thesis, DataFrames from Python's pandas and geopandas libraries will be used. Since they are broadly used data structures [59] that offer flexible and intuitive storing and handling of data in tabular form, including spatial analysis, they were chosen for this thesis.

Listing 3.2: Algorithm for constructing rectangles out of the extracted coordinate values

```
Input data: results list

results1 = first half(results)
results2 = second half(results)
combined_results = merge (results1, results2) on 'uuid'

rectangles = []
for rows in combined_results
    xmin, ymin, xmax, ymax = row[1], row[2], row[3], row[4]
    rectangle = Polygon([(xmin, ymin), (xmin, ymax), (xmax, ymax)
    , (xmax, ymin)])
    add rectangle to rectangles

store rectangles as a dataframe
Output data: rectangles dataframe
```

The second step is adding information about the CRS the extracted coordinates are in, in this case EPSG:25832, better known as ETRS89/UTM zone 32N. This information can either be added manually, or also extracted from the DB through a Cypher query.

3.2.2 OSM Data

Extraction

As previously mentioned in Section 2.4, the OSM data used in this thesis is downloaded using GEOFABRIK. They offer OSM data for set regions like Oberbayern in the form of shape files sorted into

- **point feature layers**, which include the POI layer, but also the places, places of worship, natural features, traffic related and transport infrastructure layers,
- **line feature layers**, which include the roads and paths, the railways and the waterways layer,
- **polygon feature layers**, which include the building layer, as well as the land use and bodies of water layers.

In this thesis, only the building layer and the POI layer are used, as they offer the most information about buildings. The building layer contains detailed footprints in the form of Polygons but often lacks additional details, as shown in Table 3.1. To enhance the data set, the POI layer is added, with every POI at least including feature class information and most also providing a name (see Table 3.2).

Table 3.1: Data included in the OSM building file

ID	geometry	code	fclass	name	type
120853828	(692...	1500	building	-	house
81645904	(691...	1500	building	Angerhof	-
79619583	(690...	1500	building	-	office
93979508	(691...	1500	building	-	retail
265939140	(689...	1500	building	-	-

Table 3.2: Data included in the OSM POI file

ID	geometry	code	fclass	name
60013073	(691...	2004	post_box	-
83102025	(692...	2101	pharmacy	Rumford Apotheke
251874282	(692...	2301	restaurant	Shoya
340005581	(689...	2602	atm	Postbank
409497642	(690...	2305	bar	Kauz

Preprocessing

After accessing the relevant OSM data, three preparative measures have to be taken to ensure compatibility with the extracted CityGML data:

- **The OSM data has to be in the same data format.** To be able to run operations on both data sets together, they need to be compatible with each other and the programs used for their analysis. Accordingly, the data is stored as a pandas DataFrame.
- **The spatial data has to be in the same CRS.** For any kind of spatial operation between two sets of data, they need to be in the same CRS. To integrate the data into the CityGML GraphDB at the end, it needs to be in the same CRS as the coordinates stored in the DB

(ETRS89/UTM zone 32N). To this end, it is more logical to transform the OSM coordinates from WGS84 to ETRS89/UTM zone 32N, than the other way around.

- **The spatial data has to cover the same area.** Since data from GEOFABRIK can only be downloaded for preset areas, these sets can include large amounts of data outside the scope of the area covered by the CityGML data. To avoid working with oversized data sets, which would significantly slow down later operations, the OSM data gets clipped to only features with coordinates inside the MBR that contains all CityGML coordinates. This Bbox information can either be retrieved from the GraphDB, or can be constructed by finding the outmost coordinate in every cardinal direction in the CityGML data set.

3.3 Spatial Matching

The goal of this section is to have a list of spatially matched OSM buildings and POIs, that can subsequently be integrated into the CityGML GraphDB. More specifically, the building DataFrame (Table 3.1) and the POI DataFrame (Table 3.2) should be expanded by an additional column, indicating what CityGML building(s) they are matched to, thus giving a clear overview over all the matches. A possible result for the expanded DataFrames can be seen in the tables below (Table 3.3, Table 3.4).

Table 3.3: OSM building DataFrame expanded by a column for the corresponding Bbox

ID	geometry	code	fclass	name	type	*Bbox*
120853828	(692...	1500	building	-	house	*(692...*
81645904	(691...	1500	building	Angerhof	-	*(691...*
79619583	(690...	1500	building	-	office	*(690...*
93979508	(691...	1500	building	-	retail	*(690...*
265939140	(689...	1500	building	-	-	*(689...*

Table 3.4: OSM POI DataFrame expanded by a column for the corresponding Bbox

ID	geometry	code	fclass	name	*Bbox*
60013073	(691...	2004	post_box	-	*(691...*
83102025	(692...	2101	pharmacy	Rumford Apotheke	*(692...*
251874282	(692...	2301	restaurant	Shoya	*(692...*
340005581	(689...	2602	atm	Postbank	*(689...*
409497642	(690...	2305	bar	Kauz	*(689...*

In the following, this section will initially discuss Bbox[2] to Polygon matching, then focus on POI matching, and finally give a quick overview over any objects that went unmatched.

3.3.1 Building Matches

Matching Step 1: Preliminary Matches

In this thesis, the focus of the spatial matching of buildings is on the matching between Polygons and Bboxes. Thus, some of the rules and approaches will vary from those employed in Polygon to Polygon match situations that are well documented in [6][60], but most of the same core principles are used. Spatial matching between 2D geometries is typically performed by assessing the degree of overlap between their areas [6]. To be able

[2] For the purpose of clarity and brevity, from here on, CityGML building Bboxes will be referred to as Bboxes, OSM building Polygons as Polygons and OSM POI coordinates as Points.

to analyze these overlaps, the first step is to identify all Polygons and Bboxes that intersect. Instead of using a direct method, which would be highly inefficient as it would mean comparing every single Bbox to every Polygon, an STR-tree (see Subsection 2.5.2) is used to streamline the process of querying for intersections and improve performance. The STR-tree is built from one set of geometries and then used to run search queries for intersections with the other set. This poses the question of which geometry set, Polygons or Bboxes, should be used to construct the STR-tree. Since Bboxes are rectangles, the simplest of all Polygon structures, they are computationally less expensive than more complex Polygon structures. Hence, if the STR-tree is constructed using the more complex Polygons, each query against the STR-tree (which uses Bboxes), will filter out non-intersecting Polygons quickly. This, in turn, reduces the number of computationally more expensive Polygon-Bbox intersection checks in favour of inexpensive Bbox-Bbox intersection checks. On the other hand, if the STR-tree is constructed using Bboxes, every single check will be conducted on a Bbox-Polygon pairing, which makes this version computationally a lot more expensive. Accordingly, the STR-tree should always be constructed from the more complex geometry, in this case the set of Polygons.

In addition to finding all overlapping geometry pairs, a look at the possible ways of overlap is helpful. There are three distinct possibilities, as shown in Figure 3.4:

- Bbox **covers** Polygon
- Bbox **intersects with** Polygon
- Bbox **is covered by** Polygon

Since the CityGML Bboxes represent axis-aligned MBRs for buildings, they are usually significantly larger than a Polygon representing the same building (see Figure 3.3). Therefore, the first two cases of overlap are expected for a spatial match between the two. Assuming both sets of data are of a relatively high quality, the third case only occurs, when a structure is represented as one building in the data set using polygonal geometries, but as multiple buildings in the data set using Bbox geometries. Since CityGML tends to be more detailed than OSM [6], this third case cannot be overlooked, even though building Polygons tend to be smaller than the corresponding Bboxes.

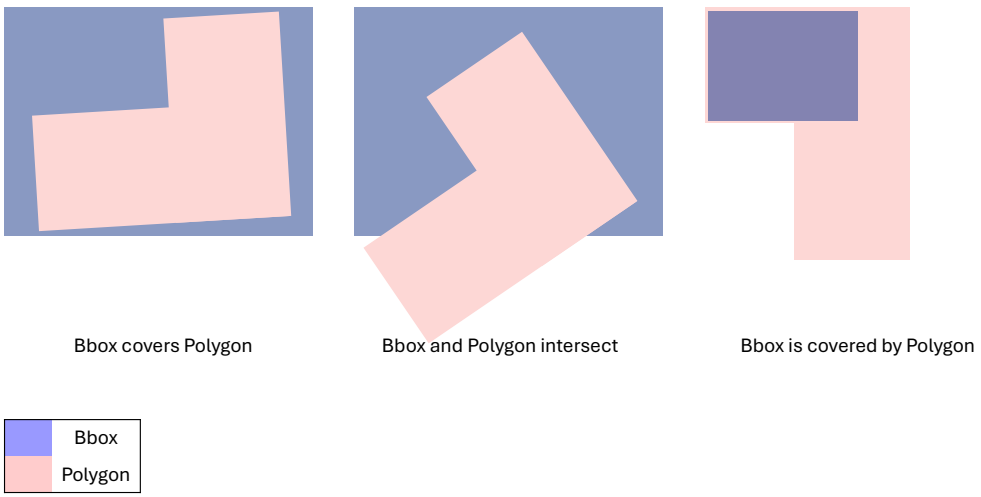


Figure 3.4: Categories of overlap between a rectangular, axis aligned Bbox and a Polygon

Matching Step 2: Accepted Matches

The last section provided a list of all overlapping geometries, as well as their type of overlap. However, the fact that a Polygon and a Bbox overlap, is not enough to accept them as a spatial match representing the same building. These intersections can stem from representing the same building, but also from scenarios like overlap due to shared walls or the geometries representing parts of the same complex, in which case a match should not be accepted (see Figure 3.5). Additionally, due to the often large nature of Bboxes in comparison to the buildings they envelope, they tend to overlap heavily, even when the actual building structures are separate, as can be seen in Figure 3.7. This leads to a significant number of intersections that should not be considered as matches.

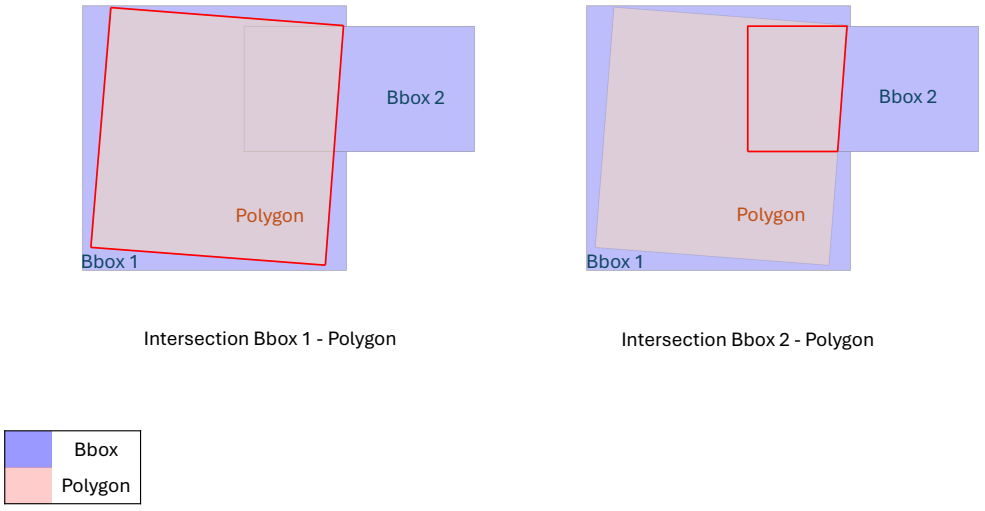


Figure 3.5: Two possible scenarios for an intersection between a Bbox and a Polygon: On the left, Bbox 1 and Polygon clearly represent the same buiding and should be accepted as a match. On the right, even though Bbox 2 and Polygon are intersecting, they do not represent the same building and should not be accepted as a match.

Following the approach of [60] and [6], the use of the subsequent equation allows to sort the degree of overlap in reference to the area of any overlapping Polygons and Bboxes:

$$\frac{|Area(Bbox) \cap Area(Polygon)|}{|min(Area(Bbox), Area(Polygon))|} \geq t \tag{3.1}$$

By dividing the area of intersection through the area of the smaller of the two geometries, the possible results range from 0, if they do not overlap at all, to 1, if one of the geometries is completely covered by the other. The result is then compared to an empirical threshold *t*, which can be adjusted based on how consistently similar both data sets are. Previous studies on this topic have found a minimum threshold of 30 % [60] to 50 % [6] to be necessary. They were, however, working with Polygon-Polygon relations, whereas here Polygon-Bbox relations are regarded. Since the Bbox geometries tend to be larger than a Polygon representing the same building, and thus often cover large parts of or even the entire Polygon, the threshold for this thesis can be set at 50 % or possibly even higher during the implementation. Additionally, for every match that was marked under the overlap categories "covers" or "covered by", the result does not need to be calculated, as it is automatically *1 = t*.

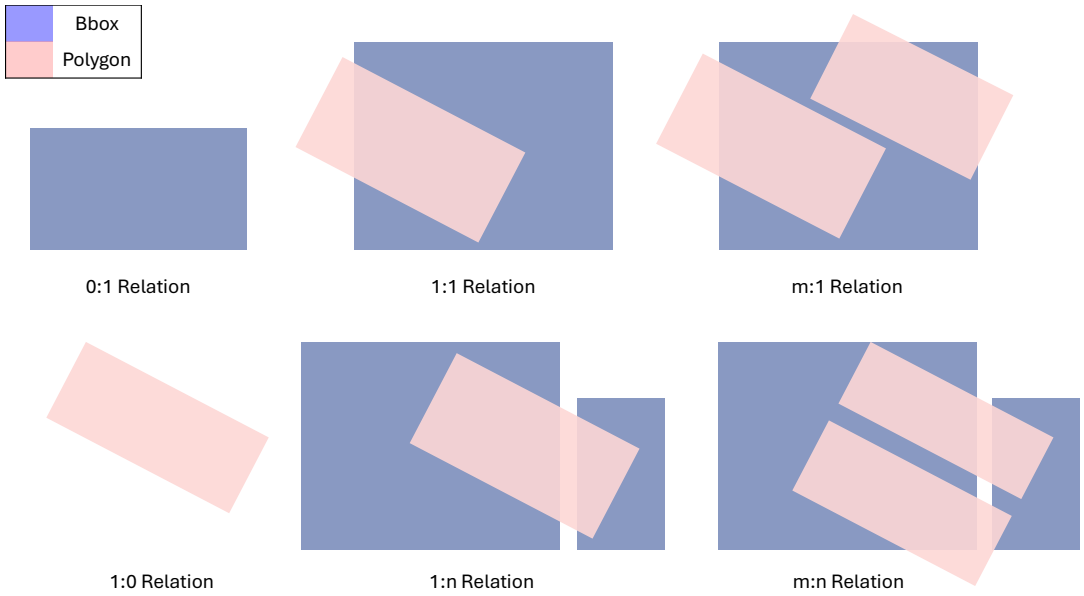


Figure 3.6: The six possible spatial matching relations by ratio: 0:1 Match, a Bbox without any matches (upper left); 1:1 Match, one Polygon matches with one Bbox (upper middle); m:1 Match, multiple Polygons match with one Bbox (upper right); 1:0 Match, a Polygon without any matches (lower left); 1:n Match, one Polygon matches with multiple Bboxes (lower middle); m:1 Match, multiple Polygons match with multiple Bboxes (lower right).

According to [60][6], there are now six cases based on the matching ratio between the two geometry sets, that need to be considered and treated separately, of which four are the possible matching scenarios and two are the possible scenarios for unmatched buildings.

0:1 Relation A Bbox without any matching Polygons, a case further discussed in Subsection 3.3.3. This includes all Bboxes that do not clear the threshold with any intersecting Polygons. Additionally, this encompasses all Bboxes that never overlapped with any Polygons. These unmatched objects can be found by comparing the original list of Bboxes extracted from the GraphDB to the list of matched Bboxes.

1:0 Relation A Polygon without any matching Bboxes, likewise discussed in Subsection 3.3.3. Like the 0:1 scenario, this includes all Polygons that do not clear the threshold as well as all Polygons with no overlap. These Polygons can also be found by comparing the initial and matched lists.

1:1 Relation In the most straightforward case, one Polygon uniquely matches with a single Bbox. This is typical for free standing buildings, and if the threshold t is surpassed, can be accepted as a definite match.

m:1 Relation Multiple Polygons matched to a single Bbox often occur when a structure is represented by several building Polygons but only one building Bbox. If the threshold is met, all Polygons can be accepted as definite matches, as they are all linked to the building represented by the Bbox and can provide it with relevant data. For example, if the OSM set contains entries for several stores within a shopping center and they all match the CityGML shopping center, the shopping center building will be enriched with information about all the stores it contains. Additionally, since the goal of the building match is mostly to provide a good basis for POI matching, more matched Polygons will provide more Points contained within their bounds.

1:n Relation A single Polygon matched to multiple Bboxes typically occurs when the Polygon set provides a less fragmented representation of a building complex, while the Bboxes offer a more detailed breakdown. This is common in the data sets used, as CityGML often captures finer distinctions between structures compared to OSM [6]. However, this situation is not trivial. Should the OSM data be applied to all matching CityGML buildings? Doing so could cause misidentification, such as when OSM represents a house and its garage as one structure, while CityGML separates them into two separate structures. If the OSM data is applied to both, the garage might be incorrectly labeled as a house, leading to inaccurate results. This could also skew any analysis conducted on the expanded data set, making it appear as though there are two houses instead of one. Therefore, it is problematic to assign the OSM data to multiple CityGML buildings. However, choosing only one single match could result in a mismatch or a severe misrepresentation, especially if the Polygon represents a big building complex split up in multiple Bbox. It is also important to remember that the building match is mostly a preliminary step for the POI matching, so the best course of action should be to match the buildings in a way most suited for that purpose. It is consequently necessary to further differentiate between the 1:n relation matches, before accepting or dismissing them.

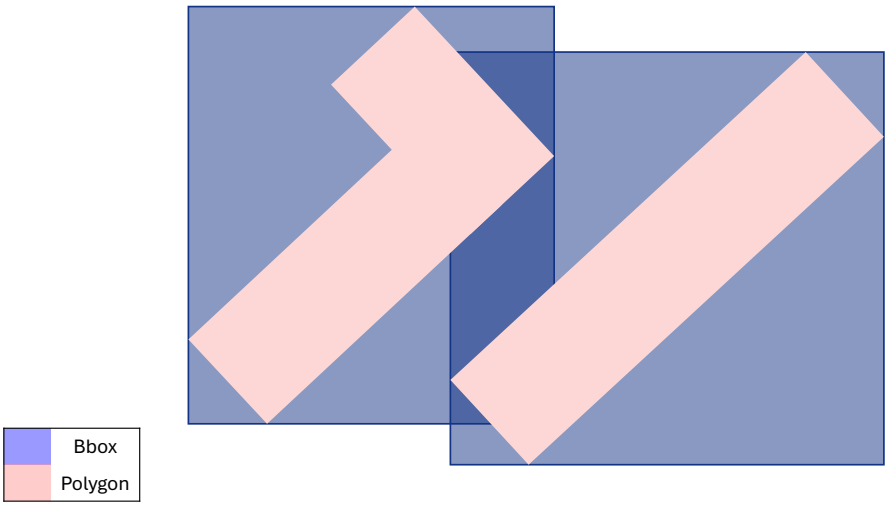


Figure 3.7: Scenario of two Bboxes overlapping. Even though two buildings can have a significant distance between them, their axis-aligned Bboxes can still overlap, creating m:n match situations.

m:n Relation Multiple Polygons matching with multiple Bboxes typically happens for the reasons mentioned above, which stem from the difference in detail between the OSM and CityGML data sets. But they could also occur because the Bboxes are expected to have a lot of overlap with other Bboxes and their respective Polygons due to their nature as often comparatively large, axis-aligned bounding rectangles, as can be seen in Figure 3.7. Like the 1:n relation matches, these matches are not trivial to solve either. However, since m:1 relations have been labelled as unproblematic in the context of this thesis, where one CityGML building having multiple OSM building matched to it is accepted, it is possible to simplify the m:n relation problem. Instead of handling a single m:n situation, it can be divided into m individual 1:n situations, which can be addressed separately. An example of such a split is given in Figure 3.8. This simplification of m:n match situations means that the amount of unclear situations is reduced to just regarding the case of 1:n ratios.

To decide how to treat the still unclear case of 1:n ratio, it is vital to look at the possible reasons behind this matching situation. As mentioned above, the most obvious reason is a differently detailed representation, where one OSM structure is represented as several CityGML structures. In this case, it would be the best

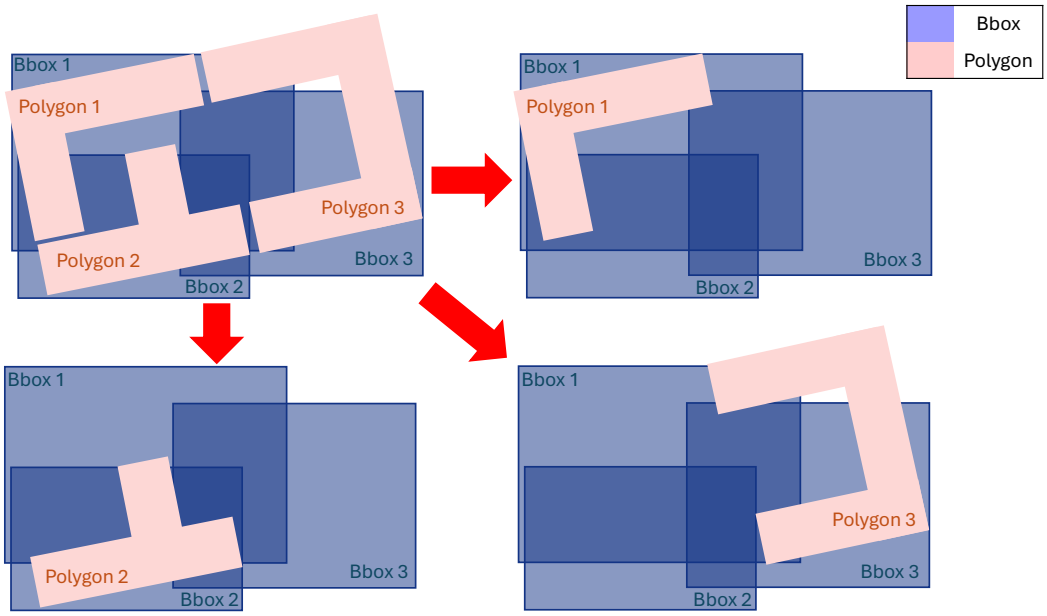


Figure 3.8: The 3:3 Match on the upper left is split into three individual 1:3 Matches, each of which can be handled independently from the other two.

course of action to match the Polygon to all Bboxes that clear the threshold *t*. Mainly because it guarantees that any POIs attached to the Polygon can later be matched to the correct CityGML "sub"structure in the larger OSM building structure.

The other possible reason stems again mostly from the nature of the CityGML Bboxes and can be viewed in Figure 3.8: Bbox 1 and Polygon 2 overlap severely, as do Bbox 2 and Polygon 2, despite the fact that only one of these combinations, namely Polygon 2 and Bbox 2 constitutes a correct match. The reason for this being that the more complex a building, the more its Bbox tends to differ in size and cover areas nowhere near its building. In this case, there is usually one single Bbox to Polygon match for each Polygon, that can be considered the correct one, thus a single match is the preferred solution in this situation.

Since both situations that lead to 1:n ratio situations require different outcomes, the question poses itself, if there is an easy way to differentiate between both cases. The solution here lies in the size difference between Polygons and Bboxes. If a Polygon and a Bbox represent the same building, the Bbox is expected to be about the same size or bigger than the Polygon (see Figure 3.3). Conversely, it can be assumed that if the Polygon is significantly larger than the corresponding Bbox, it also represents a larger building structure than the Bbox. This observation makes it possible to formulate the following rule:

Listing 3.3: Algorithm for 1:n matches

```
1   Input: Ap = Area(Polygon), Ab = Area(Bbox)
2   Output: accepted or dismissed match
3    f = buffer parameter < 1
4
5   If Ap/Ab < f
6        smallerArea = Ap
7    Else if Ap/Ab > 1/f
8        smallerArea = Ab
9    Else
10       smallerArea = -    # Both are approximately the same size
11
12   If largerArea = Ap
13       Accept the match
```

```
14      Else
15          Calculate Jaccard-Index J(Ab, Ap)
16          If J(Ab, Ap) = max(J(-, Ap))
17              Accept the match
18          Else
19              Dismiss the match
```

This rule accepts all matches, where the Polygon is determined to have a distinctly larger area, regulated through the empirical parameter *f*. Additionally, if the Polygon is not the larger area, the Jaccard-Index for the match is calculated and compared to all other Jaccard-Indices for that Polygon. Then, only the combination with the biggest Index is chosen as a match for that Polygon, assuring it is only matched to one single bigger or same-sized Bbox and all other matches for that Polygon are dismissed. By calculating the intersection of two areas over their union, the Jaccard-Index filters out the match with the most similar geometries in the context of size and location:

$$J(Bbox, Polygon) = \frac{|Area(Bbox) \cap Area(Polygon)|}{|Area(Bbox) \cup Area(Polygon)|} \tag{3.2}$$

After this last step, the Polygon-Bbox matching is concluded, with every potential match from Subsubsection 3.3.1 either having been dismissed or accepted. A full overview over all the steps can be found in Figure 3.12. It should be briefly mentioned why the Jaccard Index was not used in place of Equation 3.1. Since the Jaccard Index filters by similarity, it decreases as the size difference between the two areas increases. However, as previously mentioned, Bboxes can be many times larger than the corresponding Polygon (see Figure 3.3), which would make it impossible to define a meaningful threshold for the results returned by the Index.

3.3.2 Point Matches

Matching Step 1: Preliminary Matches

While POIs can be located anywhere and provide information about a wide range of objects, this thesis focuses solely on those that offer direct information about buildings or objects situated inside or on buildings. For this reason, only POIs within or on buildings will be considered for matching purposes. Although incorporating points from outside buildings, as demonstrated by [22], could be useful for tasks such as identifying the nearest object (e.g., determining which house is closest to a bench or trashcan), the aim here is to enrich the building itself with relevant data, focusing only on information that directly pertains to that specific building.

Instead of directly matching points to Bboxes, the following approach first matches points to Polygons, and then later utilizes the information from Subsection 3.3.1 about which Bboxes these Polygons correspond to. There are several reasons for this approach:

- Polygons are much closer representations of the actual building structure compared to Bboxes. Bboxes, as previously mentioned, are often significantly larger than the building footprint and can extend arbitrarily far from the building itself. For example, as shown in Figure 3.9 c), directly matching points to the Bbox would in this instance result in the left point being matched but not the right, even though both points are equidistant from the building.
- In the case regarded in this thesis, Polygons and points come from the same source, leading to greater consistency and homogeneity between the locations of the points and Polygons compared to the points and Bboxes. Thus making it more likely, that a point located in a Polygon is actually referencing the same building as the polygon.

- Polygons typically have less overlap than Bboxes, which means they provide a more precise spatial reference for matching the points, avoiding unnecessary overlaps and mismatches caused by Bbox overextension.

Rather than checking each point against every Polygon, which would be computationally expensive, an STR-tree is once more used to efficiently query whether a point is within or touching a Polygon. The STR-tree is constructed from the Polygon geometries, for the same reasons as in Subsubsection 3.3.1, because they are the more computationally complex structure compared to points. Additionally, similar to the Polygon to Bbox match, points can be presorted into categories: outside, within, and touching Polygons (Figure 3.9 a) to c)), of which only the points outside of all Polygons will be dismissed as matches during the STR-tree query.

- Points located **outside** of any Polygons carry information unrelated to buildings, such as locations of park benches, trash bins or tourist information signs in the case of OSM POIs.
- Points located **within** Polygons can generally be assumed to provide information relevant to the building, especially when both points and Polygons come from the same data source. These POIs can represent a range of objects found inside a building, from small entities like shops or restaurants to amenities like public restrooms or ATMs or even provide information on the building as a whole.
- **Touching** points, on the other hand, are typically positioned on or near the edges of buildings and could provide information about objects attached to or directly on the wall, like POIs representing vending machines or security cameras.

There is also the possibility that a point could match multiple Polygons if those Polygons overlap or touch (seeFigure 3.9 d) and e)). The frequency of occurrence for such points depends on the degree of overlap in the Polygon data.

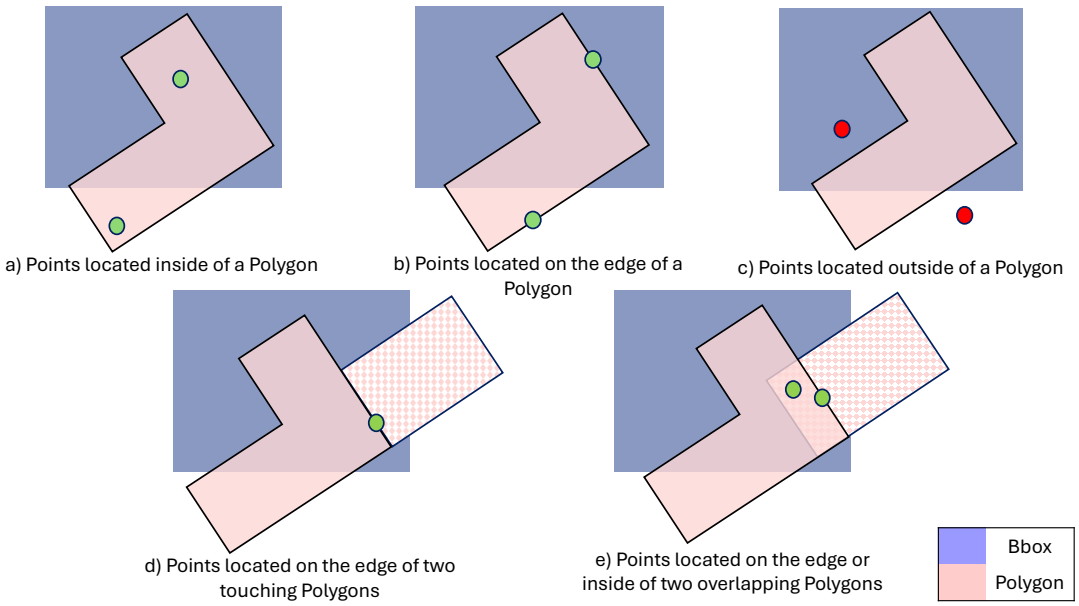


Figure 3.9: All possible cases of Point locations in regard to building Polygons: Points can be uniquely located inside a) or on the edge of a Polygon b), they can be located on the edge or inside of multiple touching d) or overlapping Polygons e) and lastly, they can be located outside of any Polygon c).

Matching Step 2: Accepted Matches

After completing the previous section, a list of all point-to-Polygon matches is now available for each Polygon that corresponds to at least one Bbox. Ideally, every point on this list should be accepted and its information integrated, as all these points are either inside or on the edge of already matched Polygons. However, how the points should be matched to the Bboxes still needs to be addressed. In cases where the **Polygon** is only matched to a **single** Bbox, the points can be automatically assigned to that Bbox. The situation becomes more complex when the **Polygon** matches **multiple** Bboxes. In such cases, it is necessary to determine which of the matched Bboxes contains the point. The following scenarios illustrate the possibilities (Figure 3.10):

- **Single Bbox match:** If the point is inside or touching just one Bbox (Figure 3.10 b)), the point can be assigned to that Bbox without issue.
- **Multiple Bbox match:** If the point is inside or touching two or more bbox (Figure 3.10 c)), the point should be assigned to all relevant bboxes. Ignoring the point would lead to significant information loss, especially given the overlapping nature of Bboxes. Assigning it to only one would carry a high risk of incorrect matching, as the arbitrary nature of Bbox sizes offers no clear indicator of proximity, unlike with Polygons.
- **No Bbox match:** If the point lies within or on the edge of a Polygon but does not match any Bbox (Figure 3.10 c)), there are two options: discard the point or assign it to the closest Bbox that matched the Polygon. The distance can be measured from either the center or the edge of the Bbox. Both approaches are valid, and it's difficult to determine which would yield better results since the outcome depends on the specific scenario. Alternatively, the distance can be measured from both the center and the edge, and the point can be assigned only if both measurements indicate the same Bbox.

All these steps can be summarised through the following algorithm:

Listing 3.4: Algorithm for point to bbox matching

```
Input: points, polygons, bboxes
Output: points matched to appropriate bboxes

for each polygon in polygons:
    get the list of points associated with the polygon
    get the list of bboxes associated with the polygon

    if polygon is only matched to one bbox:
        for each point in the list of points:
            assign the point to the single bbox

    else if polygon is matched to multiple bboxes:
        for each point in the list of points:
            find the bboxes that the point is inside or touching

            if point is inside or touches only one bbox:
                assign the point to that specific bbox

            else if point is inside or touches multiple bboxes:
                assign the point to all overlapping bboxes

            else if point does not match any bbox:
                calculate the distance from the point to each
    matched bbox
```

```
                if distance from center and distance from edge
    match:
                    assign the point to the closest bbox
                else:
                    handle conflicting distance measurements
    according to criteria (e.g., prefer center or edge or dismiss
     the match)

return matched points and their corresponding bboxes
```

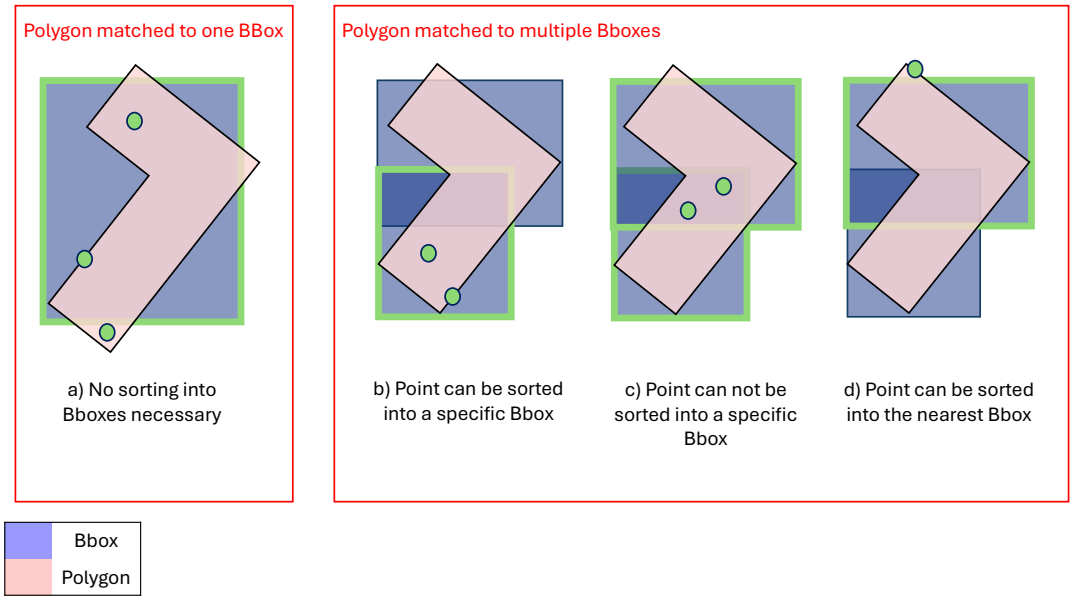


Figure 3.10: All possible cases of location for Points matched to Polygons in regard to building Bboxes: A uniquely matched Polygon, where all Points can be matched directly to the corresponding Bbox a), a Polygon that was not uniquely matched, where the Points can either be located in one b), multiple c), or none d) of the corresponding Bboxes.

3.3.3 Unmatched Buildings

When comparing the initial lists of OSM and CityGML buildings with the lists of all matched buildings, it becomes clear that some remain unmatched, the so called 1:0 and 0:1 matches. This list of unmatched buildings can mostly be broken down into two major categories (see Figure 3.11):

- **Difference in age:** When comparing two sets of data, often one can be more up to date than the other. In the case of this thesis, the used OSM data set is always less than 24 hours old at the time of download [51], while the CityGML data is from 2015 (later data was not available at [61]). This leads to the existence of new buildings in the newer (OSM) set of data, which were not built yet when the older set (CityGML) was created. Additionally, some buildings can only be found in the older data set, because they have been demolished since then. If this were the only reason for unmatched buildings, this unmatched buildings list could be used as an indicator for buildings in the CityGML data that don't exist anymore, as well as indicate, how many new buildings are missing from the CityGML data set.
- **Difference in definition:** Since there is no universal definition on what exactly should be considered a building structure or built structure, when working with two sets of data from different sources, they can have varying definitions. This concerns especially small, semi-

permanent or special structures including, but not exclusive to, bus stop shelters, statues and memorials, sheds for bicycles and garbage as well as news and concession stands. Often, these structures are considered buildings by one of the data sets, but not the other, leading to a list of structures in both building data sets that could possibly be seen as buildings, depending on one's definition. This makes it very hard to tell if a building present in the older data set does not exist anymore or is simply not defined as such by the newer data set and vice versa.

It should be noted that the disputed building structures all tend to be on the smaller side, where the lines of what can be considered a building become unclear. Consequently, it can be assumed to a certain degree that the bigger an unmatched building is, the more likely it is to fall under the category of age difference, but there are no definite ways to differentiate between the two categories.

Figure 3.11: Two examples of unmatched buildings from Munich: On the left is the construction project 'Schwanthalerstraße' showcasing the problem of time difference, with demolished buildings still existing in the CityGML data set, and newly constructed buildings only showing up in the OSM data set. On the right is a section of the 'Viktualienmarkt', highlighting the problem of a lacking universal definition. In the CityGML data set, a bus stop shelter is considered a building structure but the concession stands are not, and the OSM data set handles the same structures the exact opposite way.

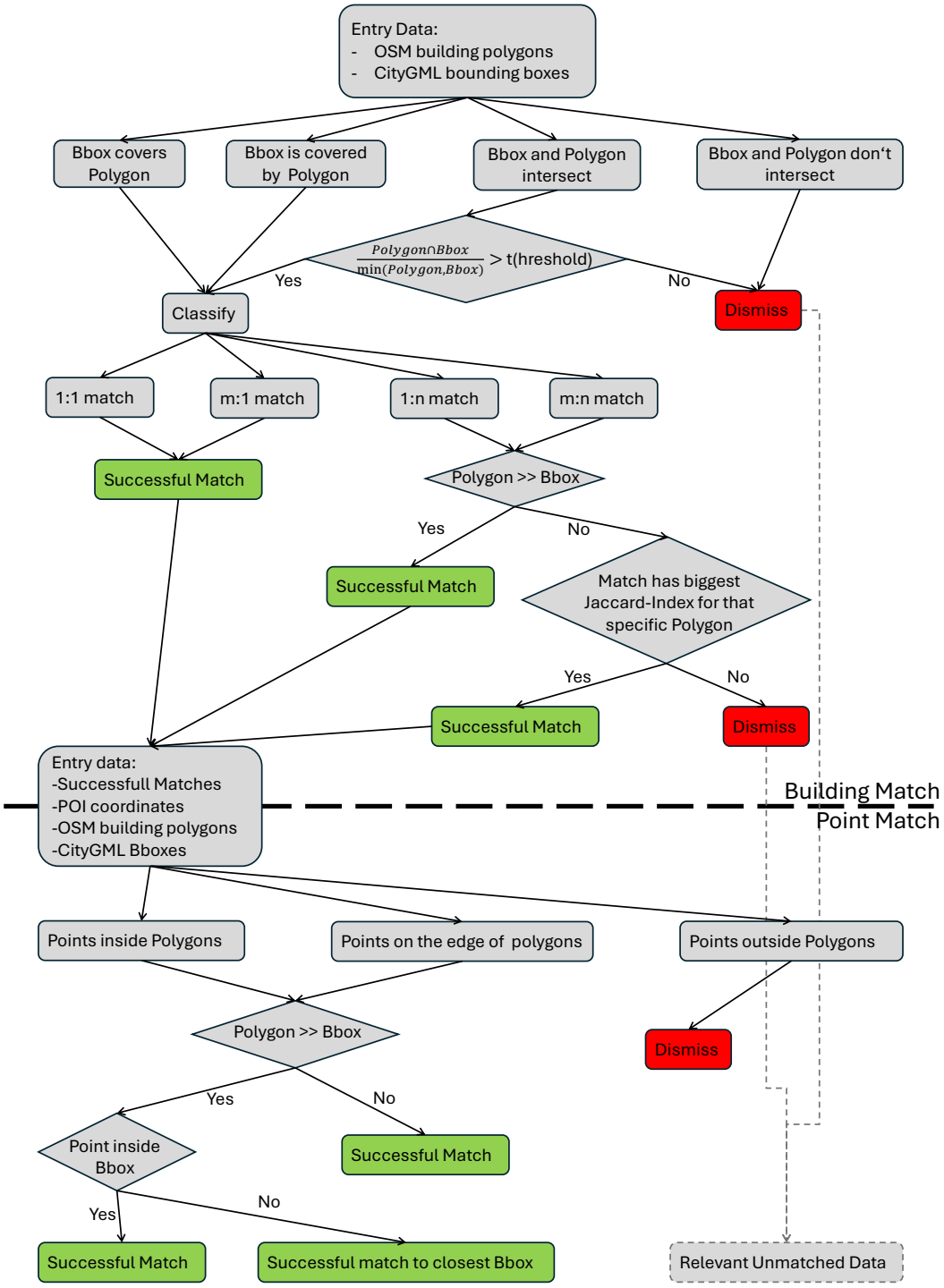
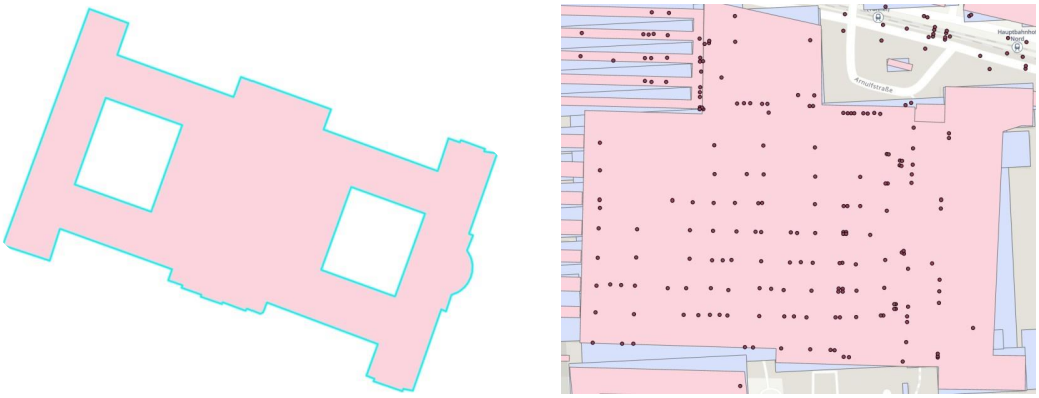


Figure 3.12: Flowchart showing the complete process spatial matching. The flowchart is split in two, the first part displays the matching of buildings through Bboxes and Polygons, the second part shows the matching of Points to Bboxes by way of using the Polygons.

3.4 Expansion of the Knowledge Graph

3.4.1 Integrating Matched Data

In this section the successfully matched data is integrated into the CityGML KG. For this purpose it is important to first identify all the information that should be integrated. As can be seen in Table 3.1 and Table 3.2, both OSM buildings and POIs provide information about their ID and geometry as well as code, feature class and sometimes name and type for buildings and code, feature class and often name for POIs, respectively. Of these, codes, feature classes, names and types provide new semantic information for the graph and consequently should be integrated into the graph. The ID can also be useful to integrate as it facilitates clear and easy identification of an OSM building or POI when comparing with outside sources. This could for example be relevant if a user wanted to update the expanded KG with more recent information, or make corrections to, or remove specific objects. Therefore, it is a good idea to always ensure that objects added to the KG can still be matched back to their source data. This leaves only the question if the geometry data for the OSM buildings and POIs should be integrated into the KG. Would integrating the geometry data have more advantages or disadvantages? Since the geometry aspect is very different for OSM buildings represented through Polygons and OSM POIs represented through points, both will be regarded seperately.



a) OSM building polygon represented through > 50 individual coordinate pairs

b) Over 100 POIs located within one large building

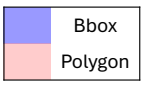


Figure 3.13: Advantages and disadvantages of storing geometries in the database: Storing building geometries, especially detailed footprints requires a significant amount of storage space. While a point coordinate can be stored in just two attribute values and a Bbox can be stored in just four, building Polygons require significantly more storage space, as can be seen in a), where a building Polygon has over 50 coordinate points and would consequently need to be stored in over 100 attribute values. On the other hand, storing POI geometries is much cheaper, since they are represented through points, and very useful, especially in large buildings and building complexes as can be seen in b). Since these points often reference objects and places inside the building and not the building as such, a loss of location information could significantly reduce the usefulness of a POI.

- **Building geometries:** As described in Figure 3.13 a), storing detailed Polygon geometries requires significant storage space. The storing of these geometries is also not uniform, as the amount of coordinate points used to represent a building varies from four (in a rectangle) to 50 and even higher. What would possible advantages of storing the geometry be? For one, as demonstrated in this thesis, spatial data can be used to uniquely identify a building.

However, since the building is already matched to a CityGML building and storing the ID for outside identification is cheaper, this advantage becomes redundant.

- **POI geometries:** Contrary to storing building Polygons, storing POI point geometries is significantly cheaper and completely uniform (see Figure 3.13), as they always provides exactly one coordinate pair. Also contrary to the building Polygons, the POI location information can be extremely relevant to a user. As can be seen in Figure 3.13 b), POIs often provide localised information about a specific part of or an object inside a building, helping the user to orient themselves inside a building. To name a practical example, if the POI only provided the information that platform 16 does exist at the train station, but not where that platform is located, the information loses much of its value.

In conclusion, it was decided that in the scope of this thesis, only the POI geometries will be integrated into the KG. It is, however, of course, possible to integrate the building geometries as coordinate values or, less costly, as a string, and this should be done if it provides additional value for the user.

After deciding on what data to integrate, the next step is to decide what the expanded graph should look like, before finally developing a method on how to integrate the new data. In order to provide consistency inside the expanded KG, it was decided that the new, added substructures should be as similarly built as possible to the preexisting graph or at least similar enough not to cause confusion. These similarities pertain to the labelling of nodes, the thematic grouping of attributes into nodes, the labelling and direction of relationships as well as the overall structuring of the new data.

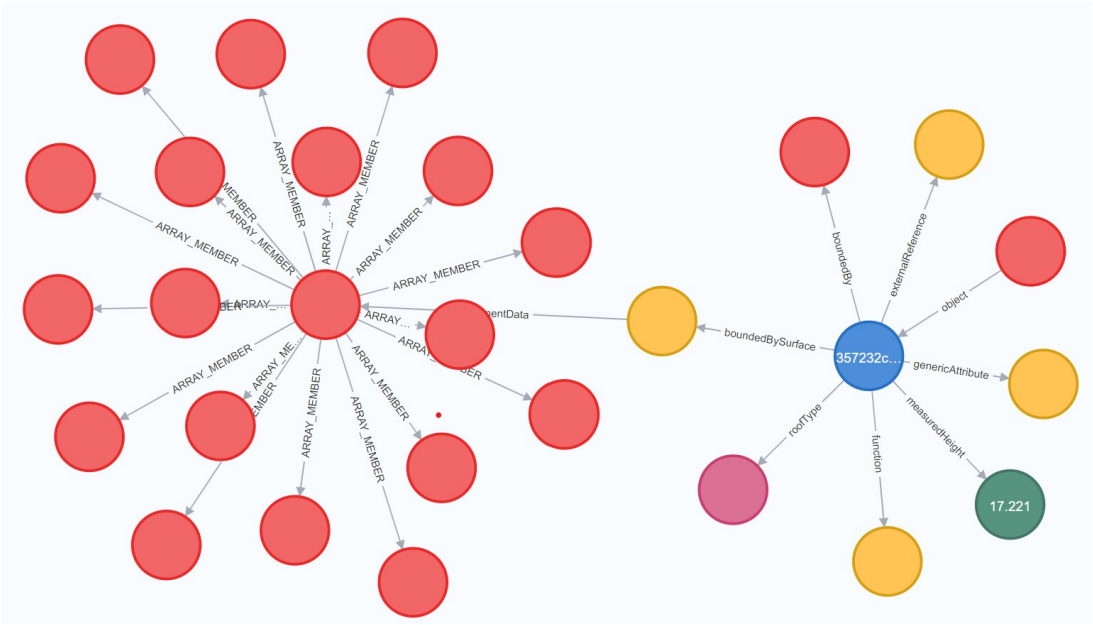


Figure 3.14: Structure of the original CityGML KG around a building node (blue): Edges are labelled uniquely for different types of node relationships (measuredHeight for the building height, roofType for the roof type...) but identically for nodes relationships of the same type (ARRAY_MEMBER for all nodes referencing bounding surfaces). Additionally, they are typically directed away from the building node if the data that is referenced belongs to it and are only directed towards the building if the building is referenced as a part of something itself, like as an object in the city model. Nodes are sorted thematically (function, bounding surfaces, roof type, external reference...) and if there is a group of nodes belonging to the same type, they are not directly connected to the central building node, but instead through a path that includes at least one other node, thus improving clearness and query performance.

With the help of the observations made in Figure 3.14, the following rules are established:

- Every OSM building and POI has its own subgraph, including all its semantic (and spatial) data.
- Data that can thematically be grouped together, like building code and building feature class, is stored in the same node.
- All new nodes have OSM in their label, so that they can be easily identified and filtered. This makes it more practical to find all nodes added through the expansion and is especially useful if the user later wants to remove the expansion.
- An additional reference node is created, connecting OSM building and POI subgraphs to the CityGML building node. Since there can be multiple nodes of the same type connected to one CityGML building, a middle node referencing the OSM_data is added. This is especially necessary in cases like Figure 3.13 b).
- All edges are directed away from the CityGML building node, since all the information in the nodes is an enrichment of that building node.
- All edges get a short name describing the relationship between their two nodes and all relationships of the same type are labelled the same.
- For better visualisation and to reduce storage, any subgraphs of OSM POIs and buildings that are matched to multiple CityGML buildings are only created once and then connected to every corresponding CityGML building through its respective OSM_data node.

A visual representation of the new subgraphs is given in Figure 3.15.

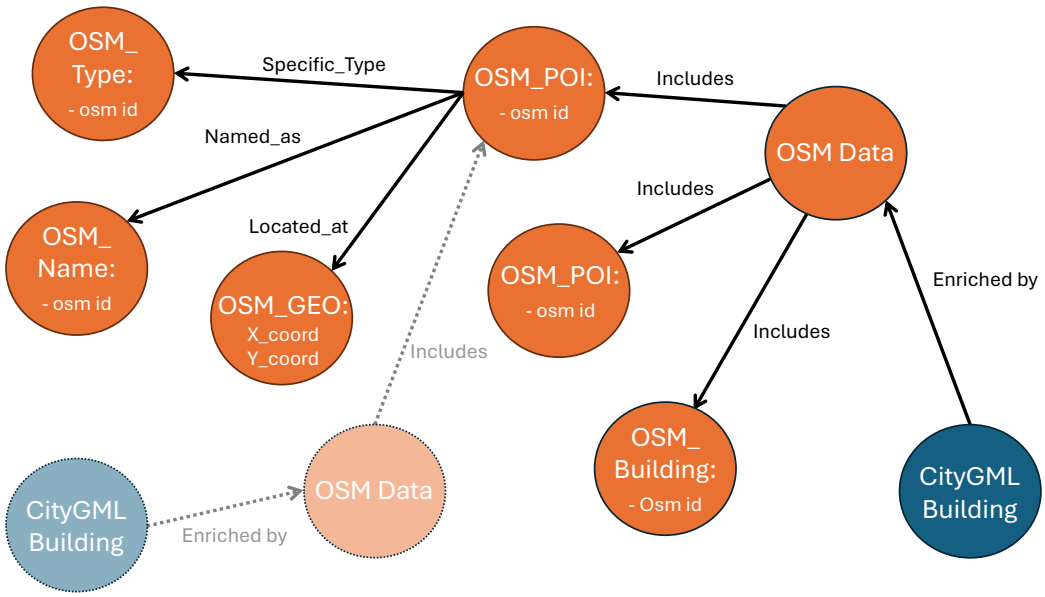


Figure 3.15: Structure of the graph expansion: The new subgraphs are introduced in keeping with the established rules to ensure consistency.

After deciding on what the expanded graph should look like, it needs to be constructed. This is done sequentially for every match by first creating the OSM_data node for the CityGML building, then creating the building or POI node and connecting it to the data node, and subsequently creating all nodes carrying information about this building or POI and connecting them. There are two important aspects to note during this construction process. First, to ensure that all OSM data is matched to the correct CityGML building, the UUID of each matched building is identified through its unique geometry and then extracted. This is

necessary to identify which building all new OSM data should be connected to. As a side note, this could also be done by avoiding the UUID and just using the geometries, but would result in far longer and more complex queries and consequentially, longer query response times. Second, it is important to differentiate, when to use the CREATE and when to use the MERGE statement. MERGE is used for nodes that should only be created once, like a specific POI node containing its unique ID. This node is only created the first time it appears on the list of matches and during any further appearances, only additional relationships are created. CREATE is used for nodes that should be created multiple times in the exact same way, for example every OSM building node is connected to its own individual type node, even if multiple buildings are of the same type and have identical type nodes. A possible algorithm for creating the new building subgraphs can be seen below[3]:

Listing 3.5: Algorithm for subgraph construction

```
FUNCTION integrate_match_data(match_pois):

 INITIALIZE osm_id_tracker AS empty dictionary

 START Neo4j transaction

 FOR each building in Neo4j:
  GET building UUID and bounding box (building_bbox)

   FOR each row in match_pois:
    GET gml_geometry, osm_id, code, fclass, name, x_coord,
   y_coord

     IF building_bbox equals gml_geometry:
      IF osm_id NOT in osm_id_tracker:
       ADD osm_id to osm_id_tracker
       ADD UUID to osm_id in osm_id_tracker

       # Create osm_data and osm_building nodes
       CREATE osm_data and ENRICHES relationship to building
       CREATE osm_building node with osm_id and INCLUDE it in
   osm_data
       CREATE osm_classification, osm_type and osm_name (if
   applicable)
       RELATE them to osm_building

      ELSE:
       ADD UUID to osm_id in osm_id_tracker

 FOR each osm_id in osm_id_tracker # create missing relationships
  FOR each UUID for that osm_id
   RELATE osm_building node to osm_data on UUID

 END Neo4j transaction
```

3.4.2 Integrating Unmatched Data

This last section will give a quick overview on how to handle unmatched data. As mentioned in Subsection 3.3.3, it is in the context of the data used in this thesis not possible to definitively decide, whether an unmatched building or point from either data set is unmatched because of age or definition problems, where

[3] The construction of the POI subgraphs uses the same algorithm with the addition of a geometry node

a match partner does not exist, or because of being badly located in correspondence to its counterpart from the other data set, where a match partner is not recognised due to a mismatch.

Consequently, all unmatched OSM POIs and Buildings are not matched to any specific building, but to the city model core node, as to not lose the information provided by these objects all together. Since they do provide the same data as their matched counterparts, their subgraphs are constructed in exactly the same way, including the OSM_data node connecting the OSM data nodes with the core model. These POIs and buildings can then be reached by using the following Cypher query:

Listing 3.6: Extraction of unmatched OSM data

```
1 MATCH (cityModel.core)-[:ENRICHED_BY]->(osm_data)-[:INCLUDES]->(b
      :osm_building)
2 MATCH (cityModel.core)-[:ENRICHED_BY]->(osm_data)-[:INCLUDES]->(p
      :osm_poi)
3 RETURN p, b
```

Additionally, all unmatched CityGML buildings are marked with a doubly labelled OSM_data:no_data_found node, which makes it possible to faster identify how many and which specific buildings went unmatched. These unmatched buildings can then be identified by the following query:

Listing 3.7: Extraction of unmatched OSM data

```
1 MATCH (n:building)-[:ENRICHED_BY]->(no_data_found)
2 RETURN n
```

4 Implementation, Evaluation and Visualisation

4.1 Implementation

In this chapter the theoretical methods developed in Chapter 3 are finally implemented. For that purpose, a two kilometer by two kilometer tile around Karlsplatz (Stachus) in Munich, Germany is chosen as a test area. The CityGML data set for that area is retrieved from BayernAtlas [61], since the data is freely available, however it should be noted that it is from 2015, since later data was not available there. This test area includes 5603 CityGML buildings and was chosen for its high building density, as well as for its variety: The area offers bounding box sizes from $2m^2$ to $35878m^2$, free standing buildings as well as complex overlap situations and is overall very heterogeneous in terms of building types and sizes. This is advantageous as it provides an assortment of many different real world scenarios. There are 3904 OSM buildings and 4620 POIs in the chosen test area. The area covered by these data sets does not correspond exactly to the one covered by CityGML, as the OSM data was extracted with a buffer parameter of 20 meters, as can be seen in Figure 4.1 below, to ensure that CityGML buildings on the edge of the tile are not missing in OSM. The lower number of OSM buildings compared to CityGML can be explained through the aforementioned fact that CityGML tends to be more detailed [6]. Additionally, the number of POIs still includes all those located outside of buildings, only 2809 POIs, meaning about 61 %, are actually located in and on the edge of OSM buildings and could potentially be matched.

Figure 4.1: Test area for the implementation: ArcGIS visualisation showing all CityGML Bboxes and OSM building Polygons inside the test area.

The entire program, from data extraction and preparation to matching and integration takes an hour to run, with about half of that time spent on the integration of the new data, and the other half distributed relatively evenly between the other sections. The implementation is executed on a computer with 32 GB RAM and 8 CPU (16 logical processors). The code for the implementation as well as for the GUI introduced in Subsection 4.2.2 can be found under https://github.com/tum-gis/citygml-osm-graph.

4.1.1 Implementation Tools

The implementation for this project were carried out in Python, one of the two languages directly supported by Neo4j. The other supported language is Java. The choice of Python was driven by its robust ecosystem, flexibility, and strong support for geospatial analysis.

The following provides an overview of the key Python libraries used:

- **Neo4j Python Library:** The Neo4j Python driver was selected due to its official support by Neo4j, making it highly reliable and well-documented. This library enables smooth interaction with Neo4j databases via the Bolt protocol, allowing Cypher queries to be executed directly in Python. In the implementation, it was used to access the Neo4j database for running read queries that extract spatial data at the beginning of the program, and for both read and write queries to integrate new data into the graph at the end [48].
- **Pandas Library:** Pandas is a widely-used, open-source library for data analysis and manipulation, known for its speed and extensive support. Neo4j's ability to convert query results directly into Pandas DataFrames through use of the Neo4j Python library made it an excellent choice for handling query results. Pandas also offers efficient filtering and analysis capabilities, such as identifying entries with the same OSM_id or CityGML geometry. The Pandas data types are compatible with integration into the Neo4j graph database, making it a valuable tool for the expansion of the CityGML Graph DB [62].
- **GeoPandas Library:** GeoPandas extends the Pandas library with support for geospatial data, providing tools to handle geometry objects for spatial analysis. It integrates with the Shapely library, allowing the storage and manipulation of spatial data, such as shapefiles. GeoPandas was used to reconstruct (Bboxes) from coordinates, store them and the OSM building and POI shapefiles from GEOFABRIK as GeoDataFrames with polygon and point geometries, and thus enabled spatial analysis through Shapely [63].
- **Shapely Library:** Shapely is a key Python package for geometric operations on planar objects, and its inclusion in GeoPandas made it central to this project. All geometric and spatial operations, such as coordinate transformations[1], intersection checks between OSM polygons and CityGML Bboxes, point location analysis, and area overlap calculations, were performed using Shapely. The library also supports the creation of STR-trees, which were used for efficient spatial indexing and querying. Shapely's STR-tree additionally provides spatial operations like 'intersects', 'within', 'contains', 'overlaps', and more, and offers a default setting of 10 child nodes per tree node, which worked well for this implementation [64].

A comprehensive list of all Python libraries used during the implementation can be found under Appendix A. Additionally to Python, ArcGIS was used to visualise and review intermediate results of the implemented program.

[1] The accuracy of the coordinate transformation was tested by comparing the original and the transformed geometries in ArcGIS. No significant deviation between the two geometries was found.

4.1.2 Implementation Specific Challenges

This section includes some of the more prominent problems that arose during the implementation, mostly because real world data is not perfect, as it is subject to human and computer error. The following gives a brief overview over how each problem was handled.

- **Data uncertainty:** Data uncertainty arises when matching POIs to buildings, as the goal is to ensure that all POIs containing information about buildings are accurately matched. This requires accounting for potential human or computer errors in the placement of POIs in relation to OSM building polygons. A common issue is that many POIs are positioned close to or on the edge of buildings, often to mark entrances or objects located on or near walls. To address this, a buffer zone was implemented, allowing any POIs located within a certain distance, d, from the building to still be counted as part of it. After testing various distances, a buffer of just 20 cm was chosen. This distance captures POIs that have accidentally been placed slightly outside the building, without including many of those POIs actually representing objects just outside, such as benches or trash cans, which are often positioned close to building polygons. During testing, this 20 cm buffer included an additional 22 POIs, 19 of which were manually verified as representing objects inside the building. In contrast, a buffer of 30 cm resulted in over 100 additional POIs, but only one additional point could be confirmed as referencing something within the building. This significant difference in accuracy led to the decision to use the smaller 20 cm buffer, balancing the inclusion of relevant POIs without introducing too much noise.

- **3D location mapping:** Unlike CityGML data, the OSM data set is purely 2D and lacks any information about building heights or storeys. For OSM buildings, this isn't problematic, as they refer to the entire building, and their height can be inferred from the corresponding CityGML buildings. If height data were available, it could be used to assess the quality of matches between data sets, as matched buildings should have similar heights. However, this lack of 3D information becomes an issue for OSM POIs. These POIs reference objects and locations within buildings, but without height data, it's unclear which storey of a multi-storey building the POI is associated with. This limitation remains unresolved in this thesis, as no height or storey data is available. Nevertheless, it would be beneficial to store height information if it existed, as it could improve the accuracy of matching and provide clearer distinctions for POI locations within buildings.

- **Tile edges:** The CityGML geometries at the edges and corners do not align precisely with the boundaries of the 2 km x 2 km square tile. Instead, all buildings that intersect with the tile are included in the data set. To ensure that the corresponding OSM buildings and POIs are also included in the matching process, a buffer zone of 20 meters was implemented around the tile. Consequently, the OSM data was clipped to create a 2040 m x 2040 m tile. This 20-meter buffer was chosen empirically as it was the smallest distance that still encompassed all CityGML buildings at the tile's edges.

- **Empirical parameters:** In addition to the previously mentioned buffer distances for addressing data uncertainty and tile edges, several other empirical parameters were set during the concept development. One key parameter is the threshold t, which started at a minimum value of 0.5 as recommended by [6]. During testing, increasing the threshold to 0.6 eliminated only a few matches, and a visual inspection confirmed that those eliminated were indeed from corresponding buildings. Lowering the threshold, on the other hand, added mostly results that during visual inspection could for the most part not be considered as matches. As a result, the threshold t was set to 0.5 for the implementation, and all subsequent results are based

on this value. The second parameter is *f*, which defines when two geometries should be considered approximately the same size. For the purpose of this thesis, after a few different values were tested, *f* was set to 0.8, meaning that an OSM building is considered roughly equivalent in size to a bounding box when its size is between 0.8 to 1.25 times the size of the bounding box and vice versa.

4.1.3 Implementation Results and Evaluation

After discussing the general details, tools and challenges of the implementation, This section will give an overview over the results of running the program on the test data set and evaluate these results[2]. Out of the 5603 CityGML buildings in the database, 78.4% were found to have matched to at least one OSM building and 1436 or 25,6% had additionally matched to at least one POI.

In a first step, the OSM data downloaded for the area of Oberbayern from GEOFABRIK was reduced down from 163,875 POIs and 1,570,917 buildings, to the tile and its buffer, resulting in 4619 remaining POIs and 3904 remaining buildings to be matched to the CityGML graph. The concentration of nearly 3% of all POIs in Oberbayern within this 2x2 km tile can be attributed to the high density of POIs in city centers. Out of the POIs, 2809 were within OSM buildings, and thus potential matches during the implementation. Out of these, 92% of POIs and 84% of OSM buildings were matched and integrated into the graph. This left 8% of POIs, 16% OSM buildings and 22% of CityGML buildings unmatched. In total 4618 building to building

Table 4.1: Amount of buildings and POIs by match ratio types (Matched total refers to the amount of buildings or POIs matched, not the total amount of matches, which would be 4618 and 2836, respectively.) The missing 139 OSM buildings and 57 OSM POIs were unmatched and in the buffer zone, and subsequently not added to the database.

	CityGML to Building	**OSM buildings**	**CityGML to POI**	**OSM POIs**
Total in tile	5603 (100%)	3904 (100%)	5603 (100%)	2809 (100%)
Matched total	4391 (78%)	3270 (84%)	1436(26%)	2571 (92%)
Matched 1:1	2236 (40%)	2236 (57%)	681 (12%)	687 (25%)
Matched 1:n	1984 (35%)	637 (16%)	153 (3%)	37 (1%)
Matched m:1	120 (2%)	343 (9%)	458 (8%)	1644 (59%)
Matched m:n	51 (1%)	54 (2%)	144 (3%)	203 (7%)
Unmatched	1212 (22%)	495 (16%)	4167 (74%)	181 (8%)

matches were created, as well as 2851 building to POI matches. The distribution over the different match ratio scenarios were found to be very similar to the ones in [6], where the same test area was used, but with CityGML ground surfaces instead of Bboxes. It was found that 48% of matches were 1:1 matches, 43% were 1:n matches, 8% were m:1 matches and just 1% m:n matches[3]. On the other hand, the distribution between match ratio scenarios for OSM POI to CityGML buildings looks very different from the building to building matches. Here, only about 24% of matches were 1:1 matches, 5% were 1:n matches, over half, at 58% were m:1 matches and 13% were m:n matches. This stark difference can be explained when looking at Figure 4.2.4 and Figure 4.2.6. These two figures show that OSM POIs tend to be often grouped together, as only a quarter of all POIs located inside or on the edge of OSM buildings are the only POI corresponding to that OSM building. In return, this means that three out of four OSM POIs share a building with at least one other POI. The explanation for this observation is that POIs relating to buildings tend to reference objects or locations inside buildings with multiple functions, one POI for each function. Additionally, the average area

[2] It should be noted, that due to the constant improvements and changes to the OSM data taken from GEOFABRIK, the exact number of data points and matches can vary slightly from day to day. The reference date for the following numbers was the 9th of October 2024.

[3] The match ratios are denoted like in the concept development. 1:n refers to one OSM building matched to multiple CityGML buildings, and m:1 to one CityGML and multiple OSM objects.

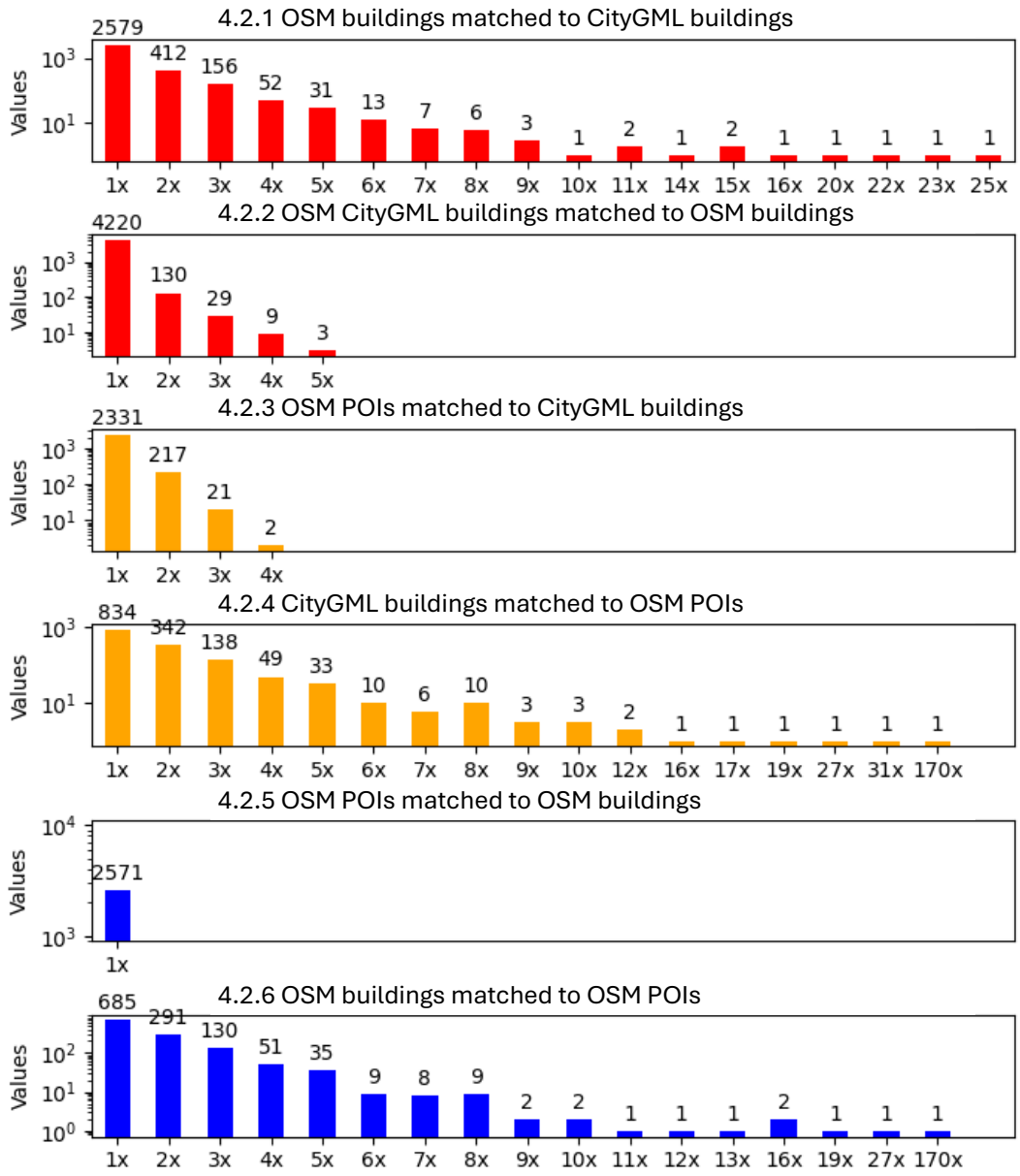


Figure 4.2: Six bar diagrams showing the match ratio distribution in each match situation. OSM building and CityGML building match (red), OSM POI and CityGML building match (orange) and OSM building and POI match (blue). The fact that all POIs are singularly matched to only one OSM building can be explained when looking at the overlap of OSM polygons: Only 0.5% of all OSM polygons overlap, making it very unlikely that a POI is located in such an overlap area. The y-axis is logarithmic for better visualisation.

of OSM buildings containing POIs was found to be about 1.5 times larger than the average OSM building area, suggesting that POIs can be mostly found in larger, more complex building structures. To give a better overview, Table 4.1 shows the distribution of CityGML buildings, OSM buildings and OSM POIs over the different possible match ratio cases, and Figure 4.2 displays the amount of match partners in every one of the three data pairings. It can be observed through all data sets, that the number of matched buildings and POIs decreases with a mostly exponential correlation to the amount of match partners, with the exception of a few distinct objects. Additionally, it can be observed, that, as suspected, OSM buildings have on average more match partners than CityGML buildings, as they are less detailed [60] and more often merge multiple buildings into one bigger building structure. Another observation is the fact that a POI is at maximum matched to four CityGML buildings, explained by the fact that all CityGML buildings sharing the same POI need to overlap with the POI coordinate. The 10 % of POIs connected to more than one CityGML building can also be seen as an indicator that the overlap between the Bboxes is very high compared to the OSM building polygons, where no POI was matched to more than one polygon. A few extreme cases can also be observed, most notably the OSM and the CityGML building matched to 170 POIs. This is the main building of Munich central station, the with 35878 m^2 by far biggest structure in the regarded tile.

Additionally to the number of matches in each ratio category, it is also interesting to regard respective area sizes in each category. As is visible in Figure 4.3, the matched total has significantly bigger areas than the tile total, both for CityGML and OSM buildings. This fits with the observation, that structures with a disputed building definition and subsequently not matchable buildings are generally on the smaller side (see Subsection 3.3.3). On top of that, there is a clear size difference between the larger Bboxes and the smaller Polygons. Furthermore, multiple match and especially m:n match situations involve larger geometries than 1:1 matches. This seems logical, given that multiple match situations usually involve bigger and more complex building structures. Lastly, the jump up in area size for CityGML in the last two match situations, which is significantly larger than any jumps in the OSM data, can be explained with the nature of Bboxes. For a big and complex building, the Bbox of that building can be multiple times larger than the building and encompass entire building block, thus almost certainly generating multiple match scenarios.

After spatial matching was completed, the matched data was integrated into the DB. The new subgraph for every unique POI and building was created first and then connected through edges to all matched CityGML building nodes. During this process, 28446 new nodes and 30059 new edges were created in the graph DB. Table 4.2 gives a brief overview over all newly created nodes and edges. As expected, the only nodes withan indegree larger than one are the OSM_building and OSM_POI nodes, as they are reachable from all their corresponding CityGML building nodes. An excerpt of the expanded KG is provided in Figure 4.4.

Table 4.2: Overview over all newly created points and edges: Every row includes a node type, the type of incoming relationship for that node and through which nodes this node type can be reached.

Node label	Amount of nodes	Incoming edge	Amount of edges	Reached through node
osm_data	4391	ENRICHED_BY	4391	cityGML.building
osm_data	1	ENRICHED_BY	1	cityGML.model
osm_building	3765	INCLUDES	5113	osm_data
osm_classification	3765	CLASSIFIED_AS	3765	osm_building
osm_name	2570	NAMED_AS	2570	osm_building
osm_type	2217	SPECIFIC_TYPE	2217	osm_building
osm_poi	2752	INCLUDES	3017	osm_data
osm_classification	2752	CLASSIFIED_AS	2752	osm_poi
osm_name	2269	NAMED_AS	2269	osm_poi
osm_geometry	2752	LOCATED_AT	2752	osm_poi
osm_data/no_data_found	1212	ENRICHED_BY	1212	cityGML.building

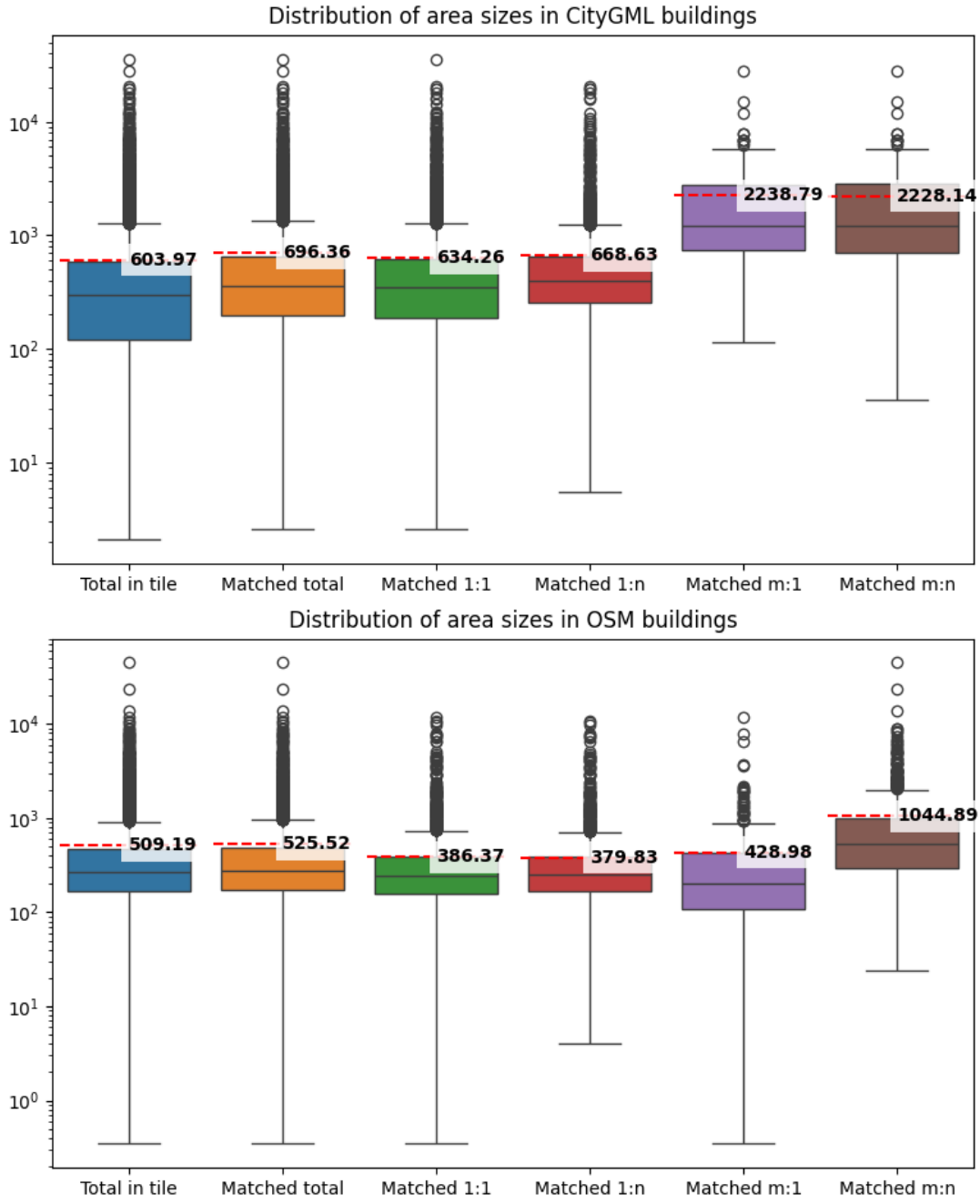


Figure 4.3: Distribution of area sizes. The upper figure shows the area distribution for all CityGML buildings that are in the tile (blue), matched to OSM buildings (orange), matched 1:1 (green), matched 1:n (red), matched m:1 (purple) or matched m:n (brown). The lower figure shows the same area distributions for OSM buildings. Additionally, all area averages are displayed through a dotted red line with their respective value on the right. The y-axis is logarithmic for better visualisation.

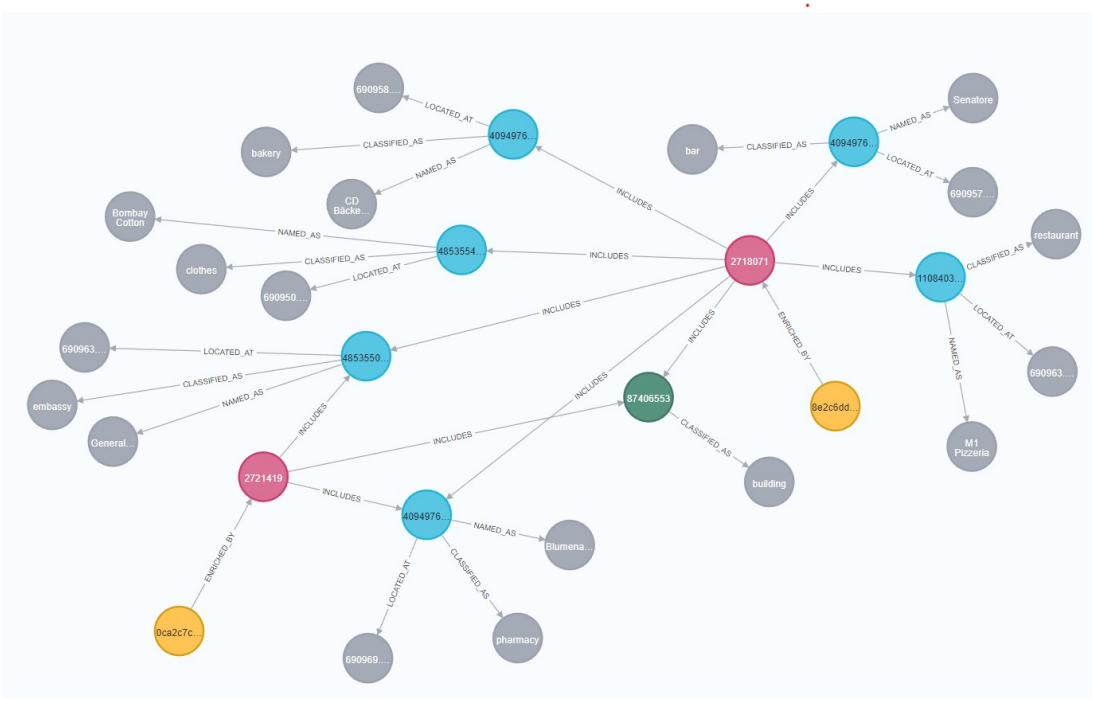


Figure 4.4: Excerpt of the expanded graph: The figure shows one of the subgraphs added during the implementation. The CityGML building nodes (yellow) from the original graph are connected to the OSM_data nodes (red), which connect the newly created subgraphs and the preexisting graph. OSM_poi nodes (blue) and OSM_building nodes (green) then make up the subgraphs together with their corresponding semantic and spatial data nodes (grey), that include for example name and function information.

This leaves the question of how good the quality of the spatial matches is. Since it is too costly to manually check all building matches, other quantitative and qualitative methods have to be employed. As a first step, evaluation methods in other papers that examine spatial matching were researched. It was found that most of these papers relied on manual checkups, either by checking randomly selected buildings [6], or employing an expert to spatially match independently from the program [22]. Additionally, there were some other methods, like the comparison of information on a specific topic present in both data sets [24]. Furthermore, approaches like intersection metrics and positional accuracy are a possibility. In the context of this thesis, it was decided to employ three different strategies for evaluating the matching process between OSM and CityGML buildings.

1. **Overall threshold:** As a first indicator, a closer look at the threshold t is taken. The threshold was chosen at 0.5, meaning that the smaller of the two intersecting geometries was at least 50% covered by the intersection. The formula for this threshold can be used to indicate the quality of matches. If the average match is only closely over the threshold, this would reflect a very weak spatial correspondence between matches in the data sets. But on the other hand, if spatial correspondence is high, the average match would have a result significantly above the threshold. The result average is calculated with the following equation:

$$\frac{sum|Area(Bbox) \cap Area(Polygon)|}{sum|min(Area(Bbox), Area(Polygon))|} \geq t \tag{4.1}$$

   The resulting average overlap for the smaller geometry is 0.86, indicating high spatial correspondence between matches. Additionally, it can be seen in the box plot in Figure 4.5, that the first and third quartiles are also very close together, with an Interquartile Range (IQR) of just 0.33 and a median at 0.95, suggesting constant high overlap.

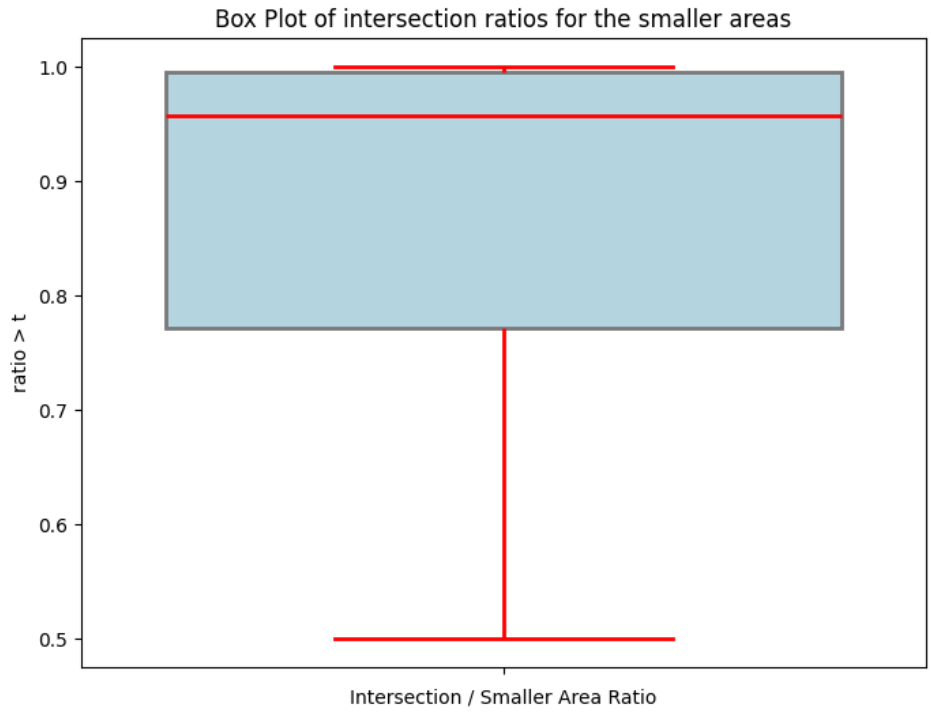


Figure 4.5: Boxplot representing the overlap/smaller area ratios of all building matches. The whiskers are set to 1.5 IQR.

2. **Random samples and confidence intervals:** As was done in most of the researched papers about spatial matching, a set of randomly selected matches, in this case 50, was reviewed for correctness. This was done by retrieving 50 matched node pairs with a Cypher query, using the RAND command. It was found that all of the 50 pairs were rightly matched, and no mismatch was identified. Furthermore, a Wilson score interval, a confidence interval that can be safely used even in the case of small sample sizes and skewed results [65], was set up to determine how high the proportion of correct matches probably is depending on those 50 samples. By choosing a 95 % (99 %) confidence interval, the Wilson score interval suggests that with a probability of 95 % (99 %), between 92.9 % (88.3 %) and 100 % (100 %) of building matches are correct.

3. **Function test:** Comparing building functions across data sets reveals notable differences and overlaps. Of the 5,603 CityGML buildings, 2,516 (45%) include function information in the form of ALKIS codes, while 2,301 out of 3,904 (59%) OSM buildings provide function information in text form. A query in the expanded database found 979 matched buildings that had functional data from both data sets. CityGML buildings were categorized into four major ALKIS function groups: residential, commercial and industrial, public, and roofing. When compared to the OSM data, which contains a much wider variety of functions (over 30 in this comparison alone), the analysis divided buildings into three categories: clear function matches, probable matches that could neither be confirmed nor dismissed, and definite mismatches. Clear matches included categories like apartments and residential, commercial and hotel, and government or hospital and public buildings. Probable matches, such as buildings labeled "elevator" or "construction" in OSM, could not be easily categorized as they may represent parts of the building or entirely different uses. The following will illustrate the

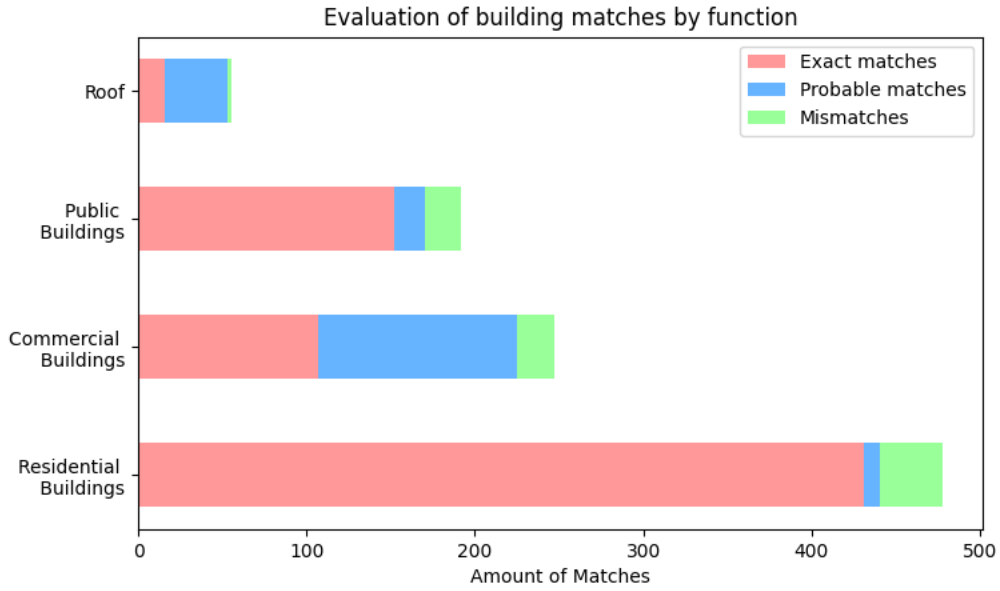


Figure 4.6: Matches classified by corresponding function: The red sections indicate complete accordance between CityGML and OSM function, blue sections indicate accordance is unclear but probable, and green sections indicate no accordance between the matched buildings.

problems encountered during the function matching. A particularly notable result is seen in the commercial building category, where many OSM buildings classified as "apartments" are marked as "commercial" in CityGML. This ambiguity often reflects buildings with mixed uses, such as a commercial ground floor with residential apartments above. Since in a data set, a building is described by a singular use, multi-function buildings are never perfectly represented and are often classified depending on which function is the most prevalent for the creator [24]. This ambiguous representation accounts for the large portion of unclear matches in the comparison, as seen in the commercial buildings section of Figure 4.6. Additionally, as seen in Table 4.1, over 21 % of matched OSM buildings and about 4 % of matched CityGML buildings are in multiple matches, making it even more difficult do deal with multi purpose building complexes. For example, a building complex represented by one OSM, but two CityGML buildings, was labelled as an office building in OSM, but as a preschool and a commercial building in CityGML, since the complex holds mainly offices, but also one Kindergarten. Another problem present in the function check was the lack of height data in OSM, resulting in multiple underground car parks being matched to an above ground CityGML building. Lastly, it should be noted, that OSM, as opposed to CityGML, defines bridges as buildings, which led to about 15 matches with bridges. In conclusion, the function test indicates that at least 73 % percent of buildings are correctly matched, providing a very conservative estimate, but cannot give a solid indication about the maximum of correct matches.

The matching between OSM POIs and CityGML buildings will not be evaluated separately. If it is assumed, that the OSM POIs are located in the correct OSM building, and the POI to CityGML matching process will be of the same quality, since it is handled through the building to building process. Testing the accuracy of correspondence between OSM POIs and OSM buildings is outside the scope of this thesis.

4.2 Querying the Database

In this section, the expanded database is accessed to showcase possible application scenarios. Several Cypher commands will be used to query the database, alongside a demonstration through a GUI that was developed to visualize the results from the expanded graph database.

4.2.1 Cypher Queries

The following provides a list of Cypher queries that could be interesting for different application scenarios, e.g. urban planning and management or tourism. They can mainly be sorted into two categories of query types, namely semantic and spatial. The first set of queries is filtering by semantic attributes, thus aiming to query the data thematically. This includes identifying all buildings with one or more certain characteristics, like a specific height, feature or location, and subsequently returning the for the user relevant data about them.

Table 4.3: Features used in queries 1-5

	CityGML	OSM	Filter
Q1	Height	Building Type	Height
Q2	UUID	Unmatched	Unmatched
Q3	Bbox	Fclass	Building Bbox
Q4	Bbox	-	Bbox area
Q5	Building Bbox	Point Name, Fclass	Bbox distance, Fclass

Q1: Find all residential buildings over 15 meters and return their height in ascending order.
Q2: Find all unmatched CityGML buildings and return their UUID data.
Q3: Find all buildings with a public toilet and return their location data.

Listing 4.1: Cypher code for Q1

```
MATCH (h:'org.citygml4j.model.gml.measures.Length')-[:
    measuredHeight]-(b:'org.citygml4j.model.citygml.building.
    Building')-[:function]-()-[:elementData]-()-[:ARRAY_MEMBER]-(
    c:'org.citygml4j.model.gml.basicTypes.Code')
WHERE c.value = '31001_1000' AND toFloat(h.value) > 15
RETURN h.value AS Building_height ORDER BY Building_height
```

Listing 4.2: Cypher code for Q2

```
MATCH (b:'org.citygml4j.model.citygml.building.Building')-[:
    ENRICHED_BY]->(no_data_found)
RETURN n.'__UUID__' AS uuid
```

Listing 4.3: Cypher code for Q3

```
MATCH (n:`org.citygml4j.model.citygml.building.Building`)-[:
    boundedBy]-()-[:envelope]-()-[:lowerCorner]-()-[:value]-()-[:
    elementData]-(d1:__ARRAY__)
MATCH (n)-[:boundedBy]-()-[:envelope]-()-[:upperCorner]-()-[:
    value]-()-[:elementData]-(d2:__ARRAY__)
MATCH (n)-[:ENRICHED_BY]->(o:osm_data)-[:INCLUDES]->(p:osm_poi)
    -[:CLASSIFIED_AS]->(c:osm_classification)
WHERE c.code = 2901     // OSM code for public toilet
```

```
5 RETURN d1.`ARRAY_MEMBER[0]` AS lower0, d1.`ARRAY_MEMBER[1]` AS
    lower1, d2.`ARRAY_MEMBER[0]` AS upper0, d2.`ARRAY_MEMBER[1]`
    AS upper1
6 // Location data is returned as (lower0, lower1, upper0, upper1)
```

Additionally to querying the data thematically, spatial filtering of the data is also possible:
Q4: Find all buildings with a Bbox area of over 20m^2 and under 30m^2 and return their area size in ascending order as well as their corresponding UUID.

Listing 4.4: Cypher code for Q4

```
1 MATCH (n:`org.citygml4j.model.citygml.building.Building`)
2     -[:boundedBy]-()-[:envelope]-()-[:lowerCorner]-()-[:value]-()
    -[:elementData]-(d1:__ARRAY__)
3 MATCH (n)-[:boundedBy]-()-[:envelope]-()-[:upperCorner]-()-[:
    value]-()-[:elementData]-(d2:__ARRAY__)
4 WITH toFloat(d1.`ARRAY_MEMBER[0]`) AS lower0,
5      toFloat(d1.`ARRAY_MEMBER[1]`) AS lower1,
6      toFloat(d2.`ARRAY_MEMBER[0]`) AS upper0,
7      toFloat(d2.`ARRAY_MEMBER[1]`) AS upper1,
8      n
9 WHERE (upper0 - lower0) * (upper1 - lower1) > 20 AND (upper0 -
    lower0) * (upper1 - lower1) < 30
10 RETURN (upper0 - lower0) * (upper1 - lower1) AS area, n.`__UUID__
    ` AS UUID
11 ORDER BY area
```

On top of querying the data both semantically and spatially, it is also possible to combine the two and create significantly more complex queries. An example is given through Q5 below.
Q5: Find all cafes, pubs and restaurants less than a kilometer away from Karlstor and return their names as well as their respective distances to the Karlstor.

Listing 4.5: Cypher code for Q5

```
1 MATCH (b_ref:`org.citygml4j.model.citygml.building.Building`)-[:
    ENRICHED_BY]-()-[:INCLUDES]-(ob_ref:osm_building {osm_id: '
    52103816'})      // osm id of the Karlstor
2 MATCH (b_ref)-[:boundedBy]-()-[:envelope]-()-[:lowerCorner]-()-[:
    value]-()-[:elementData]-(c1_ref:`__ARRAY__`)
3 MATCH (b_ref)-[:boundedBy]-()-[:envelope]-()-[:upperCorner]-()-[:
    value]-()-[:elementData]-(c2_ref:`__ARRAY__`)
4
5 MATCH (b:`org.citygml4j.model.citygml.building.Building`)
6 MATCH (b)-[:boundedBy]-()-[:envelope]-()-[:lowerCorner]-()-[:
    value]-()-[:elementData]-(c1:`__ARRAY__`)
7 MATCH (b)-[:boundedBy]-()-[:envelope]-()-[:upperCorner]-()-[:
    value]-()-[:elementData]-(c2:`__ARRAY__`)
8
9 OPTIONAL MATCH (b)-[:ENRICHED_BY]-()-[:INCLUDES]-(ob:osm_building
    )
10 OPTIONAL MATCH (b)-[:ENRICHED_BY]-()-[:INCLUDES]-(op:osm_poi)-[:
    CLASSIFIED_AS]-(c:osm_classification)
11 OPTIONAL MATCH (op:osm_poi)-[:NAMED_AS]-(n:osm_name)
12
13 WITH b, c, n,
14      (toFloat(c1_ref.`ARRAY_MEMBER[1]`) + toFloat(c2_ref.`
    ARRAY_MEMBER[1]`)) / 2 AS ref_northing,
```

```
15      (toFloat(c1_ref.`ARRAY_MEMBER[0]`) + toFloat(c2_ref.`
      ARRAY_MEMBER[0]`)) / 2 AS ref_easting,
16      (toFloat(c1.`ARRAY_MEMBER[1]`) + toFloat(c2.`ARRAY_MEMBER
      [1]`)) / 2 AS b_northing,
17      (toFloat(c1.`ARRAY_MEMBER[0]`) + toFloat(c2.`ARRAY_MEMBER
      [0]`)) / 2 AS b_easting
18 // Use of the Pythagorean theorem for spatial filtering
19 WITH b,
20      sqrt((ref_easting - b_easting) * (ref_easting - b_easting) +
       (ref_northing - b_northing) * (ref_northing - b_northing))
      AS distance,
21      c.fclass AS fclass,
22      n.name AS name
23 WHERE distance < 1000 AND (c.fclass IN ['pub', 'cafe', '
      restaurant'])
24
25 RETURN b.`__UUID__` AS uuid, distance, name ORDER BY distance
```

It is worth mentioning that since all regarded spatial data is stored through a CRS that uses northing and easting, specifically ETRS89/UTM zone 32N, it is possible to directly add and subtract coordinates or use the Pythagorean theorem. If the spatial data was encoded through longitude and latitude, as it is in WGS84, other calculation methods like the Haversine formula would have to be employed.

Lastly, it is also possible to just pull information on a single building, relevant for someone researching this specific location. These results can also be filtered, depending on the user's interest. An example is given in Q6 below.
Q6: Find any possible OSM information on the main building of Munich Central Station (UUID =622524ea-c21f-4bf6-a336-2d52a18d9e3f).

Listing 4.6: Cypher code for Q6

```
1 MATCH (b:`org.citygml4j.model.citygml.building.Building` {
      __UUID__: '622524ea-c21f-4bf6-a336-2d52a18d9e3f'})-[:
      ENRICHED_BY]->(osm_data)-[*]->(related)
2 RETURN properties(related);
```

The results of all queries can be found under Appendix B.

4.2.2 GUI Visualisation

The results from a direct Cypher query are all presented in table form. Additionally, this direct method is only accessible for users with extensive knowledge on the expanded database and the language Cypher. To consequently better visualize the results, a Graphical User Interface with a selection of possible queries and filters was built. This also allows users without any knowledge of the structure of the graph or even Neo4j, Cypher or Graph DBs in general, to query the expanded KG.

The following pages give an overview over the GUI layout, its functions and its querying capabilities. It should be noted that this Interface only represents a small excerpt of all possible queries on the expanded KG to demonstrate the possibilities of running and visualising spatial, semantic and mixed queries, both simple and complex. The GUI is split into two main parts, the console to run applications and the map to display any results. The console itself is again split into three sections, each of which handle a specific application or set of applications.

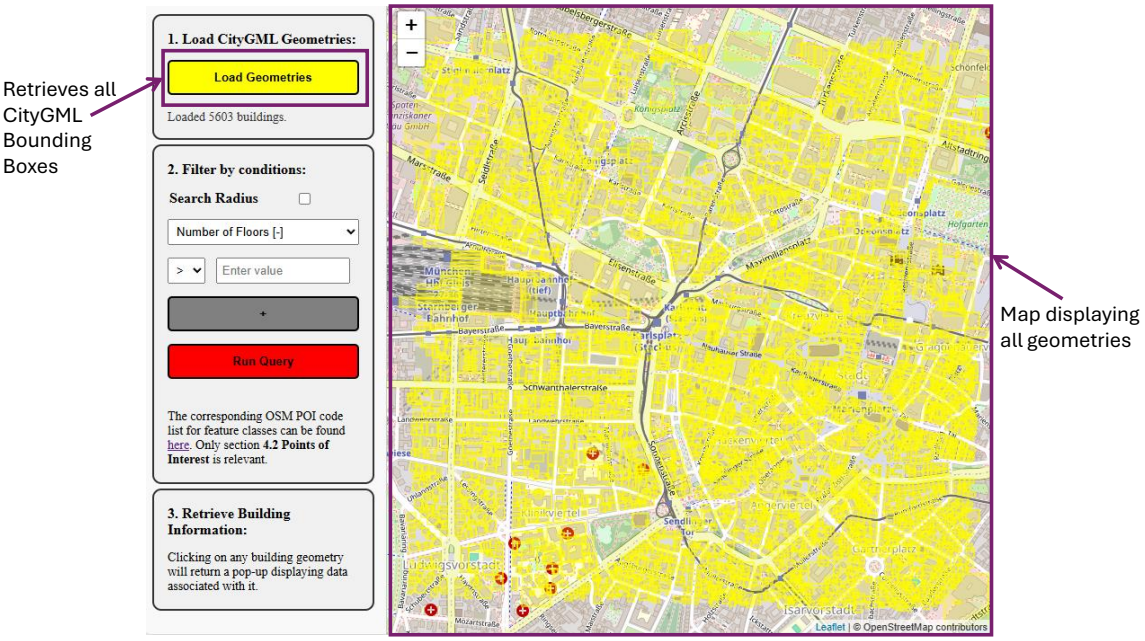


Figure 4.7: Overview over the GUI: The layout consists of a console (left) to run the queries and a map (right) to display the results. The yellow geometries are all Bboxes that are not selected by a filter.

- **Section 1: Loading Geometries** The first section of the console is demonstrated in Figure 4.7. By clicking the "Load Geometries" button, all Bbox geometries are retrieved from the KG and displayed on top of an OSM map. The total amount of geometries displayed on the map is also shown.
- **Section 2: Filtering** The second section of the console concerns the filtering of all the visualised buildings through various criteria. All available filter queries are structured the same way, namely ATTRIBUTE - COMPARISON OPERATOR - VALUE. The value is a free entry field, while the choice of attributes and comparison operators is provided via a dropdown menu (see Figure 4.8). The attributes are both spatial (e.g. area sizes) and thematic (e.g. feature class) in nature. After executing a filter query, all buildings fitting that filter are then represented through red Bboxes on the map. Additionally, all buildings where no data about the filtered attribute is available are returned in grey, all other buildings remain yellow. Both the amount of buildings matching the filter, as well as the amount of buildings not available for this filter are returned to the console.
  On top of running a single filter, it is also possible to combine two filters through an AND logic, thus combining either two thematic, two spatial or a thematic and a spatial query. An example of such a mixed filter is shown in Figure 4.8.
  Finally, it should also be possible to apply filters not just for all buildings stored in the DB, but also only for buildings in a specific area. Thus a search radius option is available, as demonstrated in Figure 4.9, where only structures within the selected radius from a specific building are filtered. The central building is hereby chosen through the user clicking on it.
- **Section 3: Specific building information** Clicking on a specific geometry will reveal a Pop Up with data about that building (see Figure 4.10). Included is data from the original graph, such as a buildings measured height, as well as data from the graph expansion, such as all feature classes and types associated with the building. This allows the user to find further information on buildings in a filter or on any specific building they might be interested in.

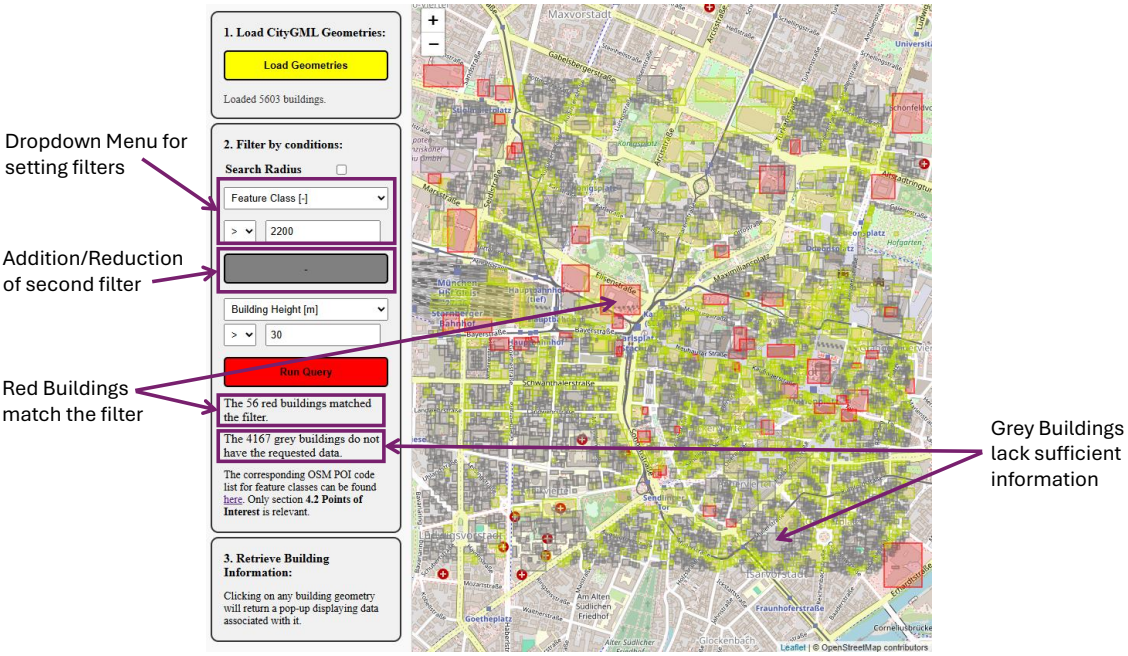


Figure 4.8: Filter query with the GUI: The query shown in this figure filters for all buildings with an associated OSM feature class with a code number > 2200 (all feature classes outside of the public and health sector [27]) AND a measured building height (CityGML) of over 30 meters. The 56 buildings matching this filter are returned in red on the map, the 4167 buildings that lack sufficient information on any of the filter criteria are returned in grey, all other buildings remain yellow.

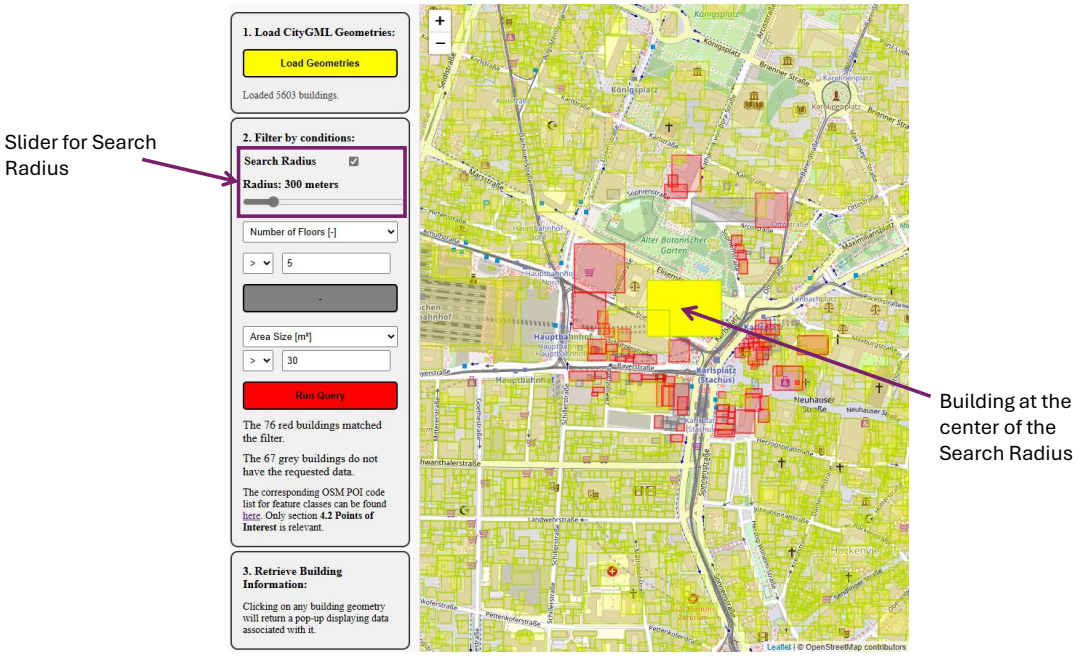


Figure 4.9: Filter query with a search radius: The filter query in this figure is only executed for buildings whose center is less than 300 meters away from the "Landgericht München" (opaque building). All buildings within the 300 meter radius are returned red, if they have more than 5 storeys above ground and an area size of over 30 m^2 and returned grey if they lack any of the filtered attributes.

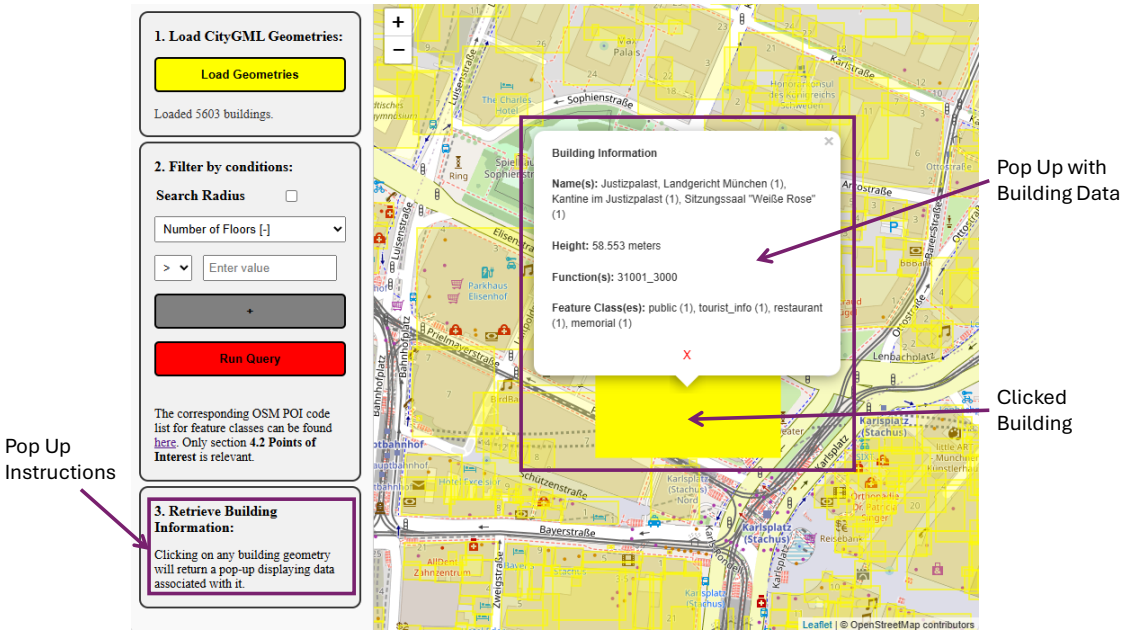


Figure 4.10: Building Pop Up: When clicking on a specific building, a Pop Up with relevant building data is shown. In this case, the Pop up displays all known feature classes and types as well as all names associated with the "Landgericht München" (OSM data), as well as the building's height and ALKIS function (CityGML data).

5 Discussion and Outlook

5.1 Conclusion

The goal of this thesis was to develop a concept allowing users without deep knowledge of CityGML to integrate thematic data into a CityGML knowledge graph and query it to meet their specific needs. A clear, ordered approach was devised and successfully implemented using OSM data, which enriched the knowledge graph. Both the concept and program are user-friendly and intuitive to query, allowing for thematic, spatial and even mixed filtering, with the GUI even enabling users without database knowledge to interpret results.

However, there are limitations. Over 20% of CityGML buildings and 15% of OSM buildings remained unmatched, suggesting that some valid matches may have been missed. These numbers could have been reduced by lowering thresholds, but that would have in turn carried the consequence of more mismatches. The approach was designed for OSM data, and it is uncertain if it would work with significantly different data sets or CityGML data from other regions [6]. Geometry matches different from bounding boxes to polygons could also pose challenges, as different spatial representations might need parameter adjustments. Reintegration of the expanded graph into CityGML is currently limited, with potential information loss. In summary, while the framework works well for expanding the knowledge graph of a data set for the purpose of analysing specific use case data - both thematically and spatially - when using OSM and the given CityGML data, further research is needed for broader applications.

5.2 Outlook

While the concept and its implementation deliver effective results, there are several areas for improvement and potential additions:

First, run time optimization is needed, particularly in the areas of data integration and point-to-polygon matching. Data integration is the most time-consuming process, so enhancing efficiency here would significantly speed up the program. Point-to-Polygon matching also takes considerably longer than Bbox-to-Polygon matching, suggesting that alternative techniques for more efficient spatial analysis might be worth exploring.

Additionally, the GUI could benefit from more advanced querying capabilities, enabling a broader range of queries, including more complex ones. For example, it could allow filtering for all residential buildings on a specific street, thus combining more complex semantic and spatial filters. Additionally, depending on the use case and the data added, the GUI could offer tailored filters, displays, and visualizations. For instance, if height data is included, visual indicators for building heights could be incorporated. All these additions could make the GUI more suitable for tasks like city management.

Exploring other data-matching techniques is another possible avenue for improvement. More sophisticated spatial approaches, such as fuzzy mapping, as suggested by [22], or non-spatial approaches [25], could

prove valuable depending on the type of data being integrated. Partly spatial approaches, like the one proposed by [24], could also enhance the matching process.

Lastly, as mentioned in the conclusion, while the current implementation focuses on integrating OSM data, the same approach could be applied to incorporate other data sources, such as heating or insulation data relevant to the energy sector. This would expand the utility of the framework across different domains and use cases.

List of Figures

List of Tables

Listings

Bibliography

[1] G. Gröger and L. Plümer, „CityGML–Interoperable semantic 3D city models," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 71, pp. 12–33, 2012, Elsevier, DOI: 10.1016/j.isprsjprs.2012.04.004.

[2] G. Gröger, T. H. Kolbe, C. Nagel and K.-H. Häfele, *OGC City Geography Markup Language (CityGML) Encoding Standard*, (OGC Doc. No. 12-019), 2.0.0, Open Geospatial Consortium, p. 344, 2012. Available: https://portal.opengeospatial.org/files/?artifact_id=47842.

[3] T. H. Kolbe and A. Donaubauer, *Urban Informatics*, Springer Verlag, ch. Semantic 3D Modeling and BIM, 2021, ISBN: 978-981-15-8982-9.

[4] F. Biljecki, K. Kumar and C. Nagel, „CityGML application domain extension (ADE): overview of developments," *Open Geospatial Data, Software and Standards*, vol. 3, pp. 1–17, 2018, Springer, DOI: 10.1186/s40965-018-0055-6.

[5] T. H. Kolbe, T. Kutzner, C. S. Smyth, C. Nagel, C. Roensdorf, et al., „OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard," *OGC standard*, no. 20-010, 2021, Open Geospatial Consortium. Available: http://www.opengis.net/doc/IS/CityGML-1/3.0.

[6] L. Ding, G. Xiao, A. Pano, M. Fumagalli, D. Chen, et al., „Integrating 3D city data through knowledge graphs," *Geo-spatial Information Science*, pp. 1–20, 2024, Taylor & Francis, DOI: 10.1080/10095020.2024.2337360.

[7] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. D. Melo, et al., „Knowledge graphs," *ACM Computing Surveys (Csur)*, vol. 54, no. 4, pp. 1–37, 2021, ACM New York, NY, USA, DOI: 10.1145/3447772.

[8] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, et al., „G-CORE: A core for future graph query languages," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1421–1432, DOI: 10.1145/3183713.3190654.

[9] R. Angles and C. Gutierrez, „Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1–39, 2008, ACM New York, NY, USA, DOI: 10.1145/1322432.1322433.

[10] D. Fernandes, J. Bernardino, et al., „Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB." *DATA*, vol. 10, pp. 373–380, 2018, SCITEPRESS, DOI: 10.5220/0006910203730380.

[11] S. H. Nguyen, „Spatio-semantic Comparison of 3D City Models in CityGML using a Graph Database," Master's Thesis, Technische Universität München, Chair of Geoinformatics, 2017.

[12] S. H. Nguyen, T. H. Kolbe, A. Donaubauer and C. Beil, „Identification and Interpretation of Change Patterns in Semantic 3D City Models," in *International 3D GeoInfo Conference*, 2023, pp. 479–496.

[13] S. H. Nguyen. „GitHub repository: citymodel-compare," 2024. [Online]. Available: https://github.com/tum-gis/citymodel-compare [visited on 07/12/2024].

[14] Neo4j. „Neo4j Cypher Query Language," 2024. [Online]. Available: https://neo4j.com/product/cypher-graph-query-language/ [visited on 07/12/2024].

[15] B. S. Hitz-Gamper and M. E. Stürmer, „Daten in OpenStreetMap integrieren–ein Leitfaden für Dateninhaber," 2021, Universität Bern, Forschungsstelle Digitale Nachhaltigkeit, DOI: 10.48350/159438.

[16] Neo4j. „Neo4j Graph Database," 2024. [Online]. Available: https://neo4j.com/product/neo4j-graph-database/ [visited on 07/12/2024].

[17] O. Contributors. „OpenStreetMap," 2024. [Online]. Available: https://www.openstreetmap.org/about [visited on 07/12/2024].

[18] E. S. Malinverni, B. Naticchia, J. L. Lerma Garcia, A. Gorreja, J. Lopez Uriarte, et al., „A semantic graph database for the interoperability of 3D GIS data," *Applied Geomatics*, pp. 1–14, 2020, Springer, DOI: 10.1007/s12518-020-00334-3.

[19] J. Lee, K.-J. Li, S. Zlatanova, T. H. Kolbe, C. Nagel, et al., „OGC® IndoorGML - with Corrigendum," *OGC® Implementation Standard*, no. 14-005r4, 2015, Open Geospatial Consortium. Available: http://docs.opengeospatial.org/is/14-005r5/14-005r5.html.

[20] H. Jang, K. Yu and S. Park, „Managing 3D GIS Data for Indoor Environment Using Property Graph Database," *IEEE Access*, vol. 11, pp. 37216–37228, 2023, IEEE, DOI: 10.1109/ACCESS.2023.3266519.

[21] J. Quinn, P. Smart and C. Jones, „3D city registration and enrichment," in *Proceedings of ISPRS COST Workshop on Quality, Scale and Analysis Aspects of City Models. Lund, Sweden*, 2009. Available: https://www.isprs.org/proceedings/xxxviii/2-W11/Quinn_Smart_Jones.pdf.

[22] P. D. Smart, J. A. Quinn and C. B. Jones, „City model enrichment," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 66, no. 2, pp. 223–234, 2011, Elsevier, DOI: 10.1016/j.isprsjprs.2010.12.004.

[23] L. A. Zadeh, „Fuzzy sets," *Information and control*, vol. 8, no. 3, pp. 338–353, 1965, Elsevier, DOI: 10.1016/S0019-9958(65)90241-X.

[24] C. Kunze and R. Hecht, „Semantic enrichment of building data with volunteered geographic information to improve mappings of dwelling units and population," *Computers, Environment and Urban Systems*, vol. 53, pp. 4–18, 2015, Elsevier, DOI: 10.1016/j.compenvurbsys.2015.04.002.

[25] S. Bischof, C. Martin, A. Polleres and P. Schneider, „Collecting, integrating, enriching and republishing open city data as linked data," in *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II 14*, Springer, 2015, pp. 57–75, DOI: 10.1007/978-3-319-25010-6_4.

[26] V. Jaillot, V. Rigolle, S. Servigne, J. Samuel and G. Gesquière, „Integrating multimedia documents and time-evolving 3D city models for web visualization and navigation," *Transactions in GIS*, vol. 25, no. 3, pp. 1419–1438, 2021, Wiley Online Library, DOI: 10.1111/tgis.12734.

[27] F. Ramm, I. Names, S. Files, F. Catalogue, P. Features, et al., „OpenStreetMap data in layered GIS format," *Version 0.7.12*, vol. 7, 2014, GEOFABRIK. Available: https://www.geofabrik.de/de/data/geofabrik-osm-gis-standard-0.7.pdf.

[28] K. Thulasiraman and M. N. Swamy, *Graphs: theory and algorithms*, John Wiley & Sons, 1992, ISBN: 0-47151356-3.

[29] N. Biggs, E. K. Lloyd and R. J. Wilson, *Graph Theory, 1736-1936*, Oxford University Press, 1986, DOI: 10.1086/352170.

[30] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, et al., „Foundations of modern query languages for graph databases," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–40, 2017, ACM New York, NY, USA, DOI: 10.1145/3104031.

[31] T. H. Kolbe. „“Geoinformatik: Vorlesung 9: Graphen und kürzeste Wege”,“ July 15, 2024. Available: https://www.moodle.tum.de/pluginfile.php/5161592/mod_resource/content/0/Geoinformatik%201%20-%2009%20-%20Graphen%20%20ku%CC%88rzeste%20Wege.pdf.

[32] S. Wasserman and K. Faust, *Social network analysis: Methods and applications*. vol. 8, Cambridge university press, 1994, DOI: 10.1017/CBO9780511815478.

[33] L. Ehrlinger and W. Wöß, „Towards a definition of knowledge graphs.“ *SEMANTiCS (Posters, Demos, SuCCESS)*, vol. 48, no. 1-4, p. 2, 2016. Available: https://ceur-ws.org/Vol-1695/paper4.pdf?utm_source=gradientflow&utm_medium=newsletter&utm_campaign=issues20.

[34] P. A. Bonatti, S. Decker, A. Polleres and V. Presutti, „Knowledge graphs: New directions for knowledge representation on the semantic web (dagstuhl seminar 18371),“ in *Dagstuhl reports*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019, pp. 29–111, DOI: 10.4230/DagRep.8.9.29.

[35] I. Tiddi and S. Schlobach, „Knowledge graphs as tools for explainable machine learning: A survey,“ *Artificial Intelligence*, vol. 302, no. 103627, 2022, Elsevier, DOI: 10.1016/j.artint.2021.103627.

[36] A. Singhal et al., „Introducing the knowledge graph: things, not strings,“ *Official google blog*, vol. 5, no. 16, p. 3, 2012. Available: https://blog.google/products/search/introducing-knowledge-graph-things-not/.

[37] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, et al., „Industry-scale Knowledge Graphs: Lessons and Challenges: Five diverse technology companies show how it’s done,“ *Queue*, vol. 17, no. 2, pp. 48–75, 2019, ACM New York, NY, USA, DOI: 10.1145/3329781.3332266.

[38] S. Zhang, T. Li, S. Hui, G. Li, Y. Liang, et al., „Deep transfer learning for city-scale cellular traffic generation through urban knowledge graph,“ in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Association for Computing Machinery, 2023, pp. 4842–4851, DOI: 10.1145/3580305.3599801.

[39] H. Ledoux. „Semantic 3D city models Lesson 6.1*,“ 2024. [Online]. Available: https://3d.bk.tudelft.nl/courses/backup/geo1004/2020/data/handout6.1.pdf [visited on 09/20/2024].

[40] T. H. Kolbe, C. Nagel and J. Herreruela, „3d city database for citygml,“ *Addendum to the 3D City Database Documentation Version*, vol. 2, no. 1, 2013, Technical University Berlin Munich, Germany. Available: https://www.3dcitydb.org/3dcitydb/fileadmin/downloaddata/3DCityDB-Documentation-Addendum-v2_0_6.pdf.

[41] P. J. Pratt and J. J. Adamski, *Concepts of database management*, Course Technology Press, p. 400, 2007, ISBN: 1423901479.

[42] Neo4j. „Neo4jDocs: Transition from relational to graph database,“ 2024. [Online]. Available: https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/graphdb-vs-rdbms/ [visited on 09/20/2024].

[43] E. F. Codd, „A relational model of data for large shared data banks,“ *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970, ACM New York, NY, USA, DOI: 10.1145/362384.362685.

[44] I. Robinson, J. Webber and E. Eifrem, *Graph databases: new opportunities for connected data*, " O’Reilly Media, Inc.", pp. 8–9, 2015, ISBN: 978-1-4919-3089-2.

[45] R. G. S. Ltd. „DB-Engines Ranking,“ [Online]. Available: http://db-engines.com/en/ranking/graph+dbms.

[46] T. Ivarsson, A. Kollegger, P. Neubauer, J. Svensson and J. Webber, „The Neo4j manual v1.3,“ *USA: Neo-Technology*, 2014. Available: https://dist.neo4j.org/neo4j-manual-1.3.pdf.

[47] F. Yu. „Best Practices to Make (Very) Large Updates in Neo4j,“ 2024. [Online]. Available: https://neo4j.com/blog/nodes-2019-best-practices-to-make-large-updates-in-neo4j/ [visited on 07/19/2024].

[48] Neo4j. „Neo4j Python Driver Manual,“ 2024. [Online]. Available: https://neo4j.com/docs/python-manual/ [visited on 07/19/2024].

[49] Neo4j. „Neo4j Cypher Manual,“ 2024. [Online]. Available: https://neo4j.com/docs/cypher-manual/ [visited on 07/19/2024].

[50] S. S. Sehra, J. Singh and H. S. Rai, „A systematic study of OpenStreetMap data quality assessment,“ in *2014 11th International Conference on information technology: new generations*, IEEE, 2014, pp. 377–381, DOI: 10.1109/ITNG.2014.115.

[51] GEOFABRIK. „GEOFABRIK downloads,“ 2024. [Online]. Available: https://download.geofabrik.de/europe/germany.html [visited on 09/18/2024].

[52] A. Guttman, „R-trees: A dynamic index structure for spatial searching,“ in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57, DOI: 10.1145/602259.602266.

[53] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos and Y. Theodoridis, *R-Trees: Theory and Applications: Theory and Applications*, Springer Science & Business Media, 2006, DOI: 10.1007/978-1-84628-293-5.

[54] Researchgate. „Object Trajectory Analysis in Video Indexing and Retrieval Applications - Scientific Figure on ResearchGate.“ 2024. [Online]. Available: https://www.researchgate.net/figure/R-Tree-indexing-example-2D-visualization-a-hierarchical-dependencies-b_fig4_225542188 [visited on 09/20/2024].

[55] S. T. Leutenegger, M. A. Lopez and J. Edgington, „STR: A simple and efficient algorithm for R-tree packing,“ in *Proceedings 13th international conference on data engineering*, IEEE, 1997, pp. 497–506, DOI: 10.1109/ICDE.1997.582015.

[56] Y. Zhang, L. Fang, Z. Du, R. Liu and J. Kang, „A grid-aided and STR-Tree-based algorithm for partitioning vector data,“ in *2011 19th International Conference on Geoinformatics*, 2011, pp. 1–6, DOI: 10.1109/GeoInformatics.2011.5980718.

[57] W. User:Chire. „R-tree with standard Guttman Split,“ 2024. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=34266673 [visited on 09/20/2024].

[58] W. User:Chire. „R-tree with STR packing algorithm,“ 2024. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=34266674 [visited on 09/20/2024].

[59] Databricks. „DataFrames,“ 2024. [Online]. Available: https://www.databricks.com/de/glossary/what-are-dataframes [visited on 09/20/2024].

[60] H. Fan, A. Zipf, Q. Fu and P. Neis, „Quality assessment for building footprints data on OpenStreetMap,“ *International Journal of Geographical Information Science*, vol. 28, no. 4, pp. 700–719, 2014, Taylor & Francis, DOI: 10.1080/13658816.2013.867495.

[61] B. Vermessungsverwaltung. „BayernAtlas,“ 2024. [Online]. Available: https://atlas.bayern.de/?c=690849,5336429&z=13.673&r=0&l=atkis,https%3A%2F%2Fgeoservices.bayern.de%2Fod%2Fwms%2Fgrid%2Fv1%2Fopendatagrid%7C%7Clod2%7C%7COpendata_Auswahl_LoD2&t=ba&tid= [visited on 09/20/2024].

[62] Python. „pandas documentation,“ 2024. [Online]. Available: https://pandas.pydata.org/docs/ [visited on 09/20/2024].

[63] Python. „geopandas documentation,“ 2024. [Online]. Available: https://geopandas.org/en/stable/docs.html [visited on 09/20/2024].

[64] Python. „shapely documentation,“ 2024. [Online]. Available: https://shapely.readthedocs.io/en/stable/index.html [visited on 09/20/2024].

[65] S. Wallis, „Binomial confidence intervals and contingency tests: mathematical fundamentals and the evaluation of alternative methods,“ *Journal of quantitative linguistics*, vol. 20, no. 3, pp. 178–208, 2013, Taylor & Francis.

Appendix

A Python Libraries

Table A.1: Table of Python libraries used in the implementation

Library	Version
neo4j	5.24.0
pandas	2.2.3
geopandas	1.0.1
shapely	2.0.6
requests	2.32.3

Table A.2: Table of Python and Java Script (js) libraries used for the Graphical User Interface

Library	Version
flask	3.0.3
neo4j	5.24.0
pyproj	3.6.1
leaflet.js	1.7.1

B Cypher Query Results

	Building Height
1	"15.014"
2	"15.064"
3	"15.08"
4	"15.094"
5	"15.142"
6	"15.194"
7	"15.199"
8	"15.255"
9	"15.264"
10	"15.273"
640	"27.133"
641	"27.17"
642	"27.845"
643	"27.886"
644	"28.022"
645	"28.143"
646	"28.716"
647	"28.927"
648	"29.455"
649	"32.77"
650	"32.784"
651	"38.616"

Figure B.1: Results for Q1: The query returns 651 indexed building heights.

	UUID
1	"56e7a250-9cf5-4fbb-9d4e-bcae18f9cd06"
2	"47c37d27-410c-4c2f-9c23-56fc2cbce4be"
3	"405e81e7-7ff3-4796-bc34-0018597ce35e"
4	"022f2a97-7bc7-4704-9cb9-2a149b3883f3"
5	"b9024a7a-acb9-4e7e-9ed5-d44383a6a004"
6	"29ae1c5b-1667-4108-9bc6-e8b4ffb96a63"
7	"596bf3cc-f0ba-4611-a498-41b2bb4cd4c9"
8	"0a80f9e2-a5cf-4a6e-ac21-6e6d119dbb16"
9	"1110b053-cbc4-4dc2-84da-e4353006caa0"
10	"1fbbaa7e-4508-4a21-b001-7c65d578015a"
1203	"3cacb436-253a-4592-8f59-9f145197fc41"
1204	"45838b64-cfb9-426b-a91a-f08a37c85b74"
1205	"334d8772-9556-4e36-aaff-678005070ac4"
1206	"6603594f-105e-4a02-8ce9-10ee5f2728b2"
1207	"b248ae9c-e9a2-43d7-a00c-45331e4e555d"
1208	"21d88442-53c0-476e-bca4-457dc113d219"
1209	"85de1f91-acfb-4198-89ac-bcb9e8d19fa8"
1210	"c737b1e5-f4ea-424a-bd1f-95f864384a6b"
1211	"b662c4c9-d421-4a55-b790-c595275e0a0f"
1212	"1e876a1c-9957-4c75-bf4e-bbee84cba5f2"

Figure B.2: Results for Q2: The query returns 1212 indexed UUIDs.

	lower0	lower1	upper0	upper1
1	"691580.88"	"5335079.39"	"691646.6"	"5335140.12"
2	"691948.232"	"5334773.1"	"691976.34"	"5334799.819"
3	"690157.07"	"5334993.94"	"690387.5"	"5335149.64"
4	"690237.35"	"5335144.22"	"690313.93"	"5335207.67"
5	"691505.3"	"5334653.5"	"691524.37"	"5334677.99"
6	"691036.86"	"5334367.92"	"691055.98"	"5334382.02"
7	"690649.27"	"5335770.52"	"690724.02"	"5335852.51"
8	"690315.514"	"5335242.188"	"690331.117"	"5335259.533"
9	"691853.419"	"5335252.788"	"691923.104"	"5335299.633"
10	"691573.4"	"5334779.84"	"691661.69"	"5334880.01"
11	"691678.983"	"5334556.759"	"691691.49"	"5334569.236"
12	"691556.849"	"5334609.759"	"691594.939"	"5334647.884"
13	"691130.28"	"5334588.57"	"691141.6"	"5334615.75"
14	"690884.27"	"5335833.3"	"690986.28"	"5335900.64"

Figure B.3: Results for Q3: The query returns the location data (Bbox coordinates) of 14 buildings.

	area	UUID
1	20,00	"c1fe84e7-82b0-4ed0-abcb-ac09c711303b"
2	20,05	"5dad83ae-72e1-432e-a9fc-421d3da0e4a4"
3	20,13	"69bce095-c8f2-42d6-8892-afa157ff7f66"
4	20,24	"2afdd4ea-2e33-4566-929f-476bb822c358"
5	20,29	"74355e6b-e54d-4f84-b493-7b7935849a50"
6	20,33	"ef506220-37c9-4b2c-bfed-b3ba328029ab"
7	20,35	"c0d4dcd2-2970-4a1f-bdc1-dba828e9394d"
8	20,37	"0e13ff17-bd06-4874-b290-11df71fb285e"
9	20,38	"c4c73a78-a469-4f80-9fe3-e1adde77bfc0"
10	20,43	"d052d4a9-881c-4cbb-99b8-6ededе75fa8a"
152	28,96	"b0ce5136-1ef9-44cc-8d95-d75c431c3119"
153	28,98	"799e9c06-aaf2-4f19-977b-6474a28289e2"
154	29,03	"5adc1bd5-a272-4935-8839-dea1afaf92ba"
155	29,06	"814ffd51-cd16-490d-afc0-2163c7837dd9"
156	29,08	"977e6583-fed2-4ce9-ad49-5fb9cdbedc45"
157	29,15	"a4a4b04c-2978-4665-8d3f-48b4b7315c8d"
158	29,11	"415d8fe7-c8ca-4b96-a77a-28b36d58047c"
159	29,22	"5a470209-df7c-4c2c-9a73-2bbdb75a228d"
160	29,39	"b192c02d-4482-4e27-b85a-9c8923e96c09"
161	29,44	"617a67dd-c902-4e4a-a01b-7020e632ae16"
162	29,48	"b8668212-2ca8-496e-a6f2-2c265d1c83f8"
163	29,48	"60299ec1-9238-4359-b2cb-cc89af9b6630"
164	29,56	"613a727b-3f97-4917-b729-cd46efe8f19b"
165	29,64	"acc018c9-df37-4020-bf42-d0aeda867fc1"
166	29,7	"c920330b-28fb-4962-8315-8d3d6ccc4a63"
167	29,75	"bcf20017-110b-48fa-942b-f8392491fb85"
168	29,75	"3c99a067-bf34-4a00-80ba-1992ddd41ce0"
169	29,79	"47c37d27-410c-4c2f-9c23-56fc2cbce4be"
170	29,84	"e06c61b5-4c55-44bb-8845-aaedbcce41af"
171	29,98	"64692895-a05b-4f6d-857c-c62146920d8c"

Figure B.4: Results for Q4: The query returns the area size of 171 buildings in ascending order as well as the corresponding UUIDs.

	UUID	distance	name
1	"3f105893-b12e-4256-9adb-cf85b6519a98"	36.3	"Starbucks"
2	"84d653d4-8bdb-4634-86c9-3e796c996606"	41.2	"Macinino"
3	"11649563-1a1c-4381-ad00-1e170cdd863f"	65.8	"Café Bistro Rendezvous"
4	"9539a7b6-d3e3-4a46-8fc7-86d7d6324d49"	72.1	"TeaOne München"
5	"339cddff-b9a4-42a5-811b-b9e764ce3a82"	84.9	"Café Bistro Rendezvous"
6	"5d41183e-44af-4744-8bc9-bd351c46d882"	92.1	"Schnitzelwirt im Spatenhof"
7	"5d41183e-44af-4744-8bc9-bd351c46d882"	92.1	"Schnitzelwirt im Spatenhof"
8	"5d41183e-44af-4744-8bc9-bd351c46d882"	92.1	"Schnitzelwirt im Spatenhof"
9	"5d41183e-44af-4744-8bc9-bd351c46d882"	92.1	"Schnitzelwirt im Spatenhof"
10	"7bdcae40-a03c-4113-9684-7759ffc1c705"	97.0	"Shanghai Restaurant"
492	"6652727d-6a5b-460d-afdc-5039fdc9325c"	984.9	"ITALIA im Tal"
493	"d790b0cd-3cdd-46f0-ac3d-b073551b10bb"	986.2	"SEN"
494	"887db18c-8f71-4373-b0da-fae6221596f1"	989.2	"Café Gelateria al Teatro"
495	"e3901a88-1c74-4935-bbff-161fd9e3b5db"	992.4	"Hofbräuhaus"
496	"74ada0db-4395-4574-ba4a-15770ea756e8"	993.0	"Osteria Il Tenore"
497	"444953ad-95c2-42d6-b36d-a8e8d8f97949"	994.3	"Tini's"
498	"afef55b1-b69b-4123-b55a-dda9dfde4421"	997.9	"Caffè Leone"
499	"cab4da46-a9f9-428f-a7cf-49026d5adc59"	998.7	"Deutsche Eiche"
500	"9022d14f-d79d-4625-bb62-d8085cc4da70"	999.8	"Mandarin China Restaurant"
501	"9022d14f-d79d-4625-bb62-d8085cc4da70"	999.8	"Mandarin China Restaurant"

Figure B.5: Results for Q5: The query returns 501 indexed UUIDs, distances and names in an ascending order of distance.

Figure B.6: Results for Q6: The query returns all attributes of new OSM nodes from the graph expansion that can be reached from the specified building node.