DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Spatio-semantic Comparison of 3D City Models in CityGML using a Graph Database

Huynh Duc An Son Nguyen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Spatio-semantic Comparison of 3D City Models in CityGML using a Graph Database

Räumlich-semantischer Vergleich von 3D-Stadtmodellen in CityGML mittels einer Graphdatenbank

Author:	Huynh Duc An Son Nguyen
Supervisor:	PD Dr. rer. nat. Georg Groh
Advisors:	Univ.-Prof. Dr. rer. nat. Thomas H. Kolbe
	M. Sc. Zhihang Yao
Submission Date:	May 15, 2017

Slightly updated version with spelling and grammatical corrections.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, May 15, 2017 Huynh Duc An Son Nguyen

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my advisor, Univ.-Prof. Dr. rer. nat. Thomas H. Kolbe, for his constant guidance and support throughout the research. Without his excellent academic expertise on the topic, this work would not have been possible. I am truly grateful for his insights and, especially, for the many long discussions we had during the research. His lessons were and always are a delight, and for that I am extremely honoured. Furthermore, I would also like to express my sincerest appreciation to M. Sc. Zhihang Yao for his continuous support and encouragement over the course of this thesis. I am wholeheartedly thankful for his numerous feedbacks and constructive advices. Finally, I would like to thank PD Dr. rer. nat. Georg Groh for giving me the opportunity to conduct my research.

Abstract

A city may have multiple CityGML documents recorded at different times or surveyed by different users. To analyse the city's evolution over a given period of time, as well as to update or edit the city model without negating modifications made by other users, it is of utmost importance to first compare, detect and locate spatio-semantic changes between CityGML datasets. This is however difficult due to the fact that CityGML elements belong to a complex hierarchical structure containing multi-level deep associations, which can basically be considered as a graph. Moreover, CityGML allows multiple syntactic ways to define an object leading to syntactic ambiguities in the exchange format. Furthermore, CityGML is capable of including not only 3D urban objects' graphical appearances but also their semantic properties. Since to date, no known algorithm is capable of detecting spatio-semantic changes in CityGML documents, a frequent approach is to replace the older models completely with the newer ones, which not only costs computational resources, but also loses track of collaborative and chronological changes. Thus, this research proposes an approach capable of comparing two arbitrarily large-sized CityGML documents on both semantic and geometric level. Detected deviations are then attached to their respective sources and can easily be retrieved on demand. As a result, updating a 3D city model using this approach is much more efficient as only real changes are committed. To achieve this, the research employs a graph database as the main data structure for storing and processing CityGML datasets in three major steps: mapping, matching and updating. The mapping process transforms input CityGML documents into respective graph representations. The matching process compares these graphs and attaches edit operations on the fly. Found changes can then be executed using the Web Feature Service (WFS), the standard interface for updating geographical features across the web.

Contents

Acknowledgements **iv**

Abstract **v**

Acronyms **ix**

1 Introduction and Motivation **1**

1.1 Motivation and Problem Statement 1
1.2 Research Objectives and Questions 2
1.3 Methodology 3
1.4 Application Scenario and Employed Tools 5
1.5 Expected Results 5
1.6 Outline 6

2 Theoretical and Methodical Background **7**

2.1 City Geography Markup Language (CityGML) 7
2.2 XML Processing 10
2.2.1 XML Parsing 10
2.2.2 XML Data Binding 12
2.3 citygml4j - The Open Source Java API for CityGML 13
2.4 Graph Database in Neo4j 13
2.4.1 From Relational to Graph Database 13
2.4.2 CityGML in Graph Database 16
2.4.3 Neo4j Graph Database Management System 16
2.4.4 Graph Structures in Neo4j 18
2.4.5 Developing in Neo4j 19
2.5 R-tree Data Structure and Neo4j Spatial 23
2.5.1 R-tree Data Structure 23
2.5.2 Neo4j Spatial 26
2.6 Web Feature Service (WFS) 28
2.6.1 WFS Communications 28
2.6.2 WFS Operations 30
2.6.3 WFS for CityGML 31

3 Mapping 3D City Models in CityGML onto a Graph Database 34
3.1 Reading CityGML Datasets in Java 37
3.2 Converting Java Objects to Graph Entities 38
3.2.1 Instance-based Approach 40
3.2.2 Hierarchy-based Approach 44
3.2.3 Combination of Instance and Hierarchy-based Approach 47
3.3 Connecting Mapped City Objects using XLinks 50
3.3.1 Existence of XLinks in Mapped Graphs 50
3.3.2 Resolving XLinks within the Graph Database 50
3.4 Calculating Minimum Bounding Boxes of Mapped City Objects 52
3.4.1 Reverse-mapping Graphs to Java Objects 53
3.4.2 Calculating Minimum Bounding Boxes of Java Objects 53

4 Matching 3D City Models in CityGML using a Graph Database 54
4.1 Comparing Node Properties 54
4.2 Matching Node Relationships 56
4.2.1 Matching Geometry of Points 58
4.2.2 Matching Geometry of Line Segments 60
4.2.3 Matching Geometry of Rings 62
4.2.4 Matching Geometry of Polygons 68
4.2.5 Matching Geometry of Solids 70
4.2.6 Matching Geometry of Minimum Bounding Boxes 70
4.3 Spatial Matching Strategies 74
4.3.1 Matching in a Grid Layout 74
4.3.2 Matching in an R-tree 82

5 Updating 3D City Models in CityGML using a Graph Database 85
5.1 Edit Operations 85
5.1.1 Class Model 85
5.1.2 Practical Example 86
5.2 Updating Building Objects using WFS 91
5.2.1 WFS Transactions on Building Objects 93
5.2.2 WFS Transactions on Thematic Properties 95
5.2.3 WFS Transactions on Geometric Properties 97
5.2.4 WFS Transactions on Complex Properties 99

6 Performance Optimization 104
6.1 Memory Tuning and Storage Selections 104
6.1.1 Memory Tuning 104

6.1.2 Storage Selections . . . 106
6.2 Batch Transaction Processing . . . 107
6.3 Concurrent Processing . . . 107
6.3.1 Multi-threading Approaches . . . 107
6.3.2 Deadlock Avoidance . . . 109

7 Application Results and Discussion 111
7.1 Test Setup . . . 111
7.1.1 Testing Environment . . . 111
7.1.2 Input Data . . . 111
7.1.3 Test Configurations . . . 111
7.2 Application Results . . . 113
7.2.1 Statistics of Mapped Graph Database . . . 113
7.2.2 Single and Multi-threading Performance . . . 115
7.2.3 Indexing Performance . . . 116
7.2.4 Differences in Performance between Building Batch Sizes . . . 117
7.2.5 Performance of the Grid Layout and R-tree . . . 118
7.3 Discussion . . . 121

8 Conclusion and Future Work 122

List of Figures 124

List of Tables 126

Listings 128

Bibliography 129

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 16, 22

ADE Application Domain Extensions. 13

API Application Programming Interface. v, 10, 11, 13, 19, 21–23, 37, 38, 40, 122, 128

AWT Abstract Window Toolkit. 62, 66–68

CityGML City Geography Markup Language. iv–vi, 1–10, 12, 13, 16, 17, 25, 28, 31, 34–103, 106, 108, 111–114, 122–126, 128

CPU Central Processing Unit. 111, 121

CRS Coordinate Reference System. 59

DOM Document Object Model. 10–12, 37, 124

GC Garbage Collector. 105–107

GIS Geographic Information System. 28, 124

GML Geography Markup Language. 4, 7, 8, 10, 13, 28, 30, 31, 122

HDD Hard Disk Drive. 106

HTTP Hypertext Transfer Protocol. 29

JAXB Java Architecture for XML Binding. 12, 13, 37–39, 41, 50, 124

JTS Java Topology Suite. 26, 62, 66

JVM Java Virtual Machine. 105, 106, 111

KVP Key/Value Pair. 29, 30, 32, 93

LOD Level of Detail. 7, 8, 111, 124

OGC Open Geospatial Consortium. 1, 7, 28, 31, 95, 122

OOP Ordinary Object Pointer. 106

OS Operating System. 104

PCIe Peripheral Component Interconnect Express. 111

RAM Random Access Memory. 104–106, 111

RDBMS Relational Database Management System. 14

SAX Simple API for XML. 10–12, 37, 50

SIG3D Special Interest Group 3D. 7

SOAP Simple Object Access Protocol. 29

SQL Structured Query Language. 14, 19

SRS Spatial Reference System. 59

SSD Solid-state Drive. 106, 111

StAX Streaming API for XML. 11, 12, 37, 50, 94

UML Spatial Reference System. 5, 8, 9, 39, 86, 124, 125

URL Uniform Resource Locator. 29

WFS Web Feature Service. iv–vi, 1, 3, 5, 6, 28–33, 91, 93, 95–103, 122–124, 126, 128

WFS-T Transactional Web Feature Service. 33

xAL eXtensible Address Language. 13

XLink XML Linking Language. vi, 2, 4, 16, 34, 36, 50–53, 111, 112, 116, 121, 122

XML eXtensible Markup Language. v, 4, 7, 9–13, 16, 29, 30, 32, 34, 37–39, 41, 50, 93, 94, 97, 99, 102, 122, 125

XPath XML Path Language. 10, 96, 97, 100, 102

XSLT Extensible Stylesheet Language Transformations. 10

1 Introduction and Motivation

1.1 Motivation and Problem Statement

As an official OGC standard for encoding virtual 3D city models, CityGML opens up opportunities for applications in a broad range of areas such as urban planning, facility management, environmental simulations and thematic inquiries. One of the main factors contributing to this success is CityGML's capability of including not only 3D urban objects' graphical appearances, but also their semantic properties. This ensures CityGML documents can be shared over various applications that make use of the model's common semantic information, which is "especially important with respect to the cost-effective sustainable maintenance of 3D city models" [Grö+12].

However, although the increasing number of CityGML datasets in recent years indicates a positive sign of the open standard's steady growth, it has also been a great challenge to maintain sustainable 3D city models. One prominent example is the difficulty of handling undocumented collaborative as well as chronological changes of an existing city model, which is currently unavoidable due to the fact that as cities grow over time, so does the need to adjust their models accordingly [NBU10]. Furthermore, because the current state of CityGML does not store such changes in its instances, multiple model documents of the same city may accumulate over time. As a result, during the maintenance phase, old city datasets are overwritten completely by newer ones, which not only costs a large amount of time and computational resources, but also loses track of collaborative changes and neglects the city's progress recorded during the given time period. Moreover, replacing entire large datasets due to only some small changes would cause an unnecessarily huge volume of transactions, especially if the database is managed via the Web Feature Service (WFS).

Therefore, instead of replacing older records, an ideal alternative should first compare the models, then attach edit operations to detected deviations, based on which only real changes are committed. This way, not only is there no more need to abandon any active models, but older documents can both be updated and still retain their respective core structure, including own rules of syntax and internal object references. This plays an important role in enabling a version control system for collaborative work in modelling and storing digital 3D city models [Cha+15]. Moreover, the number of transactions required for a WFS-enabled database is also reduced significantly.

In order to achieve this, the first task is to determine key aspects, based on which CityGML models should be compared. Since "one of the most important design principles for CityGML is the coherent modelling of semantics and geometrical/topological properties" [Grö+12], a comparison between two CityGML instances should take into account both their geometrical and semantic aspects to ensure reliable results. For example, wall surfaces can be defined in-line for each building or referenced to other walls of surrounding buildings via the XML Linking Language (XLink).

Thus, the further question arises as to how geometrical and semantic information of CityGML documents can be stored and compared. Since CityGML elements belong to a complex hierarchical structure containing multi-level deep associations, which can basically be considered as a graph, all information of a CityGML instance can theoretically be stored in a graph database. Hence, two CityGML models are equivalent if, and only if, their respective graph representations are also equivalent.

Since to date, no known available algorithm is capable of detecting semantic and geometrical changes in CityGML objects [Red14], in response to the above-mentioned questions, the thesis proposes an approach that compares two CityGML models with respect to both semantic and geometrical properties using a graph database.

1.2 Research Objectives and Questions

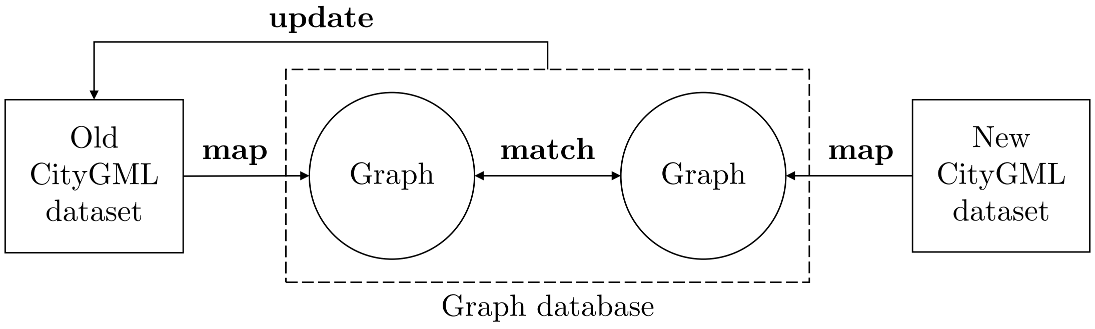


Figure 1.1: An overview of three major steps mapping, matching and updating of 3D city models using a graph database.

The main goal of this thesis is to develop a conceptual method of comparing two arbitrarily sized CityGML datasets with respect to both their semantic and geometrical information using a graph database. In general, the process can be divided into three main steps (see Figure 1.1), in each of which their respective questions are listed as guidelines below:

1. **Mapping:** Two CityGML datasets are mapped onto two corresponding graphs in the same graph database. Research questions:
    1.1. How can an arbitrarily large-sized CityGML datasets be mapped into a graph database efficiently concerning memory and computational constraints?

    1.2. CityGML objects belong to a well-defined class hierarchy. How can this information be stored in a value-based graph database such as Neo4j?

2. **Matching:** The mapped objects in graph database are matched with respect to their spatial and semantic properties. Research questions:
    2.1. An object can be defined in different syntactic ways in CityGML. How can these syntactic ambiguities be disambiguated?

    2.2. How to deal with changed identifiers of the same real world objects stored in both datasets without losing their internal structure and object references?

    2.3. How to store and compare thematic values in graph nodes?

    2.4. In which strategic pattern should objects be compared based on their geometrical properties? What are the essential spatial traits that can reduce the matching time of two objects in two and three-dimensional space?

3. **Updating:** The comparison results stored in the graph database are employed to update the older city model. Research questions:
    3.1. What should the class model of all edit operations look like?

    3.2. How to convert edit operations stored in graph database to WFS transactions?

The proposed approach restricts only to the `Building` module (with `Appearances`) defined in the CityGML schema. However, this can in principle be extended for other modules due to their schematic similarities.

1.3 Methodology

To solve the above-mentioned questions, a number of tools and algorithms are considered. Firstly, an open source graph database software such as Neo4j is chosen. Prior to the mapping step, based on the chosen graph database, a small test phase is performed to examine its compatibility with CityGML documents as well as third-party libraries, e.g. citygml4j and Neo4j Spatial. Some adjustments and fine tunings might be needed, especially for large sample datasets. However, regardless of employed software, the developed algorithms should ideally remain platform and application-independent to ensure its generality and flexibility in as many use cases as possible.

The next step is to investigate potential syntactic ambiguities allowed in GML (such as objects declared in-line or by the XML Linking Language (XLink)) according to XML specifications [Bra+08]. However, since XLink reuses existing objects by referencing to their identifiers [DeR+10], it is beneficial to take full advantage of this feature not only to accelerate mapping time by eliminating redundant objects, but also for future optimizations, especially in the matching step.

The next major part is to match two graph representations previously mapped from CityGML models. Two nodes can be matched by comparing all their property values and outgoing relationships recursively. Matching nodes simply by using their identifiers often results in a much faster runtime, as the matching of a pair of nodes reduces to a text comparison [Cha+96]. However, since a real world object may be represented by objects with different IDs from two datasets, the matching process should not rely solely on their identifiers. Thus, in such cases, to compare nodes particularly of geometric objects with potentially altered identifiers, a "walk-through" strategy is needed, since a brute-force comparison between all possible node pairs from two large graphs could cause a computationally expensive quadratic time complexity $\mathcal{O}(n^2)$. Conventional `diff` tools, such as the Hunt–McIlroy algorithm [HM76], are only applicable for pure texts and thus not compatible with hierarchical data structures like XML and its application schemata. The majority of relevant existing matching algorithms are available for XML and its tree representation, such as the tree-to-tree method proposed by [RPB09]. In addition, the algorithms [Sel77; Tai79; ZS89; Cha+96] and [CAM02] are well-known for their capability of detecting changes in ordered trees, in which it does matter how the children of an inner node are ordered [Red14]. On the other hand, unordered trees can be compared by using a heuristic "MH-Diff" algorithm in cubic time complexity in worst-case scenarios [CG97]. Later, [WDC03] suggests another method called "X-Diff" that considers tree equivalence as isomorphism.

However, despite being an application schema of XML, CityGML is not a tree data structure by definition, as it may contain cycles and nodes linked by multiple parents. Therefore, above-mentioned methods are generally not compatible with CityGML's graph data structure. Hence, this research proposes an approach specifically developed to match two CityGML graph representations. Furthermore, especially for large datasets, the ability to efficiently preselect candidates for the matching process based on their spatial properties is prioritized. Topologically relative allocations of objects can be expressed by the "4" or its extension "9-intersection model" ("4-IM" or "9-IM") [EF91; EH91]. Furthermore, an object can be localized by recursively dividing its parent graph into the so-called quadrants (2D quadtrees) or octants (3D octrees) and colouring their interior as well as exterior [Ber+08]. Alternatively, a grid layout or an R-tree can be applied to spatial objects grouped in regions based on their topological properties. Since R-trees are balanced, their query response time in logarithmic time complexity

$\mathcal{O}(\log_M n)$ is very efficient in large graphs, where M is the maximum number of entries allowed per internal node. For primitive geometries such as points, lines and polygons, their equivalence can be examined by comparing respective coordinates or shapes with error tolerances taken into account.

If two nodes are spatially matched, they may still differ on the semantic level. Therefore, after the geometric comparison is complete, a semantic matching follows. Potential equivalences based on specifications provided by [Grö+12] should be considered. For instance, a solid geometry can be further subdivided into wall and roof surfaces. Comparing such objects leads to the comparisons of their compositions as well.

The final step is to store detected deviations in the underlying graph database, so that they can easily be retrieved and processed. Such deviations in node properties or relationships are represented by edit operations, for which a UML class diagram is to be designed. These edit operations are linked to the deviation sources by graph edges, while they themselves are stored as nodes in the graph database. Then, based on the current specifications of the WFS, the standard interface for updating geographical features across the web, edit operations can be transformed to WFS transactions to update the corresponding CityGML dataset stored in a geospatial database. Note that these edit operations are not limited to the WFS and hence can also be utilized by other updating methods.

1.4 Application Scenario and Employed Tools

The main test use case of this research uses the CityGML dataset of the entire city of Berlin, which contains approximately 550,000 buildings over the area of 890 km^2 and occupies up to 20 GB of disk space. The application is implemented in Java. Unmarshalling CityGML documents is done with the help of the citygml4j library. The free-of-charge community version of the graph database Neo4j together with its plug-in Neo4j Spatial are employed. To update a WFS-enabled database, the official WFS version as well as its vendor-specific extension virtualcityWFS provided by virtualcitySYSTEMS are applied.

1.5 Expected Results

The proposed approach should be able to map reasonably large CityGML datasets into a graph database correctly. The matching step should theoretically deliver reliable results based on objects' geometrical and semantic properties efficiently. The edit operations should be available in the graph database after the matching process is complete, so that detected changes can be transformed to WFS transactions when required.

1.6 Outline

Chapter 2 introduces the theoretical and methodical background of some important concepts and tools employed during the course of this thesis. Chapter 3 describes the process of mapping CityGML instance documents to their respective graph representations stored in a graph database. Mapped graphs of two CityGML models are then compared on both geometrical and semantic level in Chapter 4. Spatial search strategies are also introduced in this chapter. CityGML documents can then be updated with the help of created edit operations described in Chapter 5, where their transformation to corresponding WFS transactions is also presented. Chapter 6 introduces some optimization possibilities, such as memory tuning and concurrent processing. Results of experiments conducted in this thesis are presented and discussed in Chapter 7. Finally, Chapter 8 concludes and mentions some possible future work of this research.

2 Theoretical and Methodical Background

2.1 City Geography Markup Language (CityGML)

Figure 2.1: Official logo of CityGML. Source: SIG3D.

"CityGML is an open data model and XML-based format for the storage and exchange of virtual 3D city models. It is an application schema for the Geography Markup Language version 3.1.1 (GML3), the extendible international standard for spatial data exchange issued by the Open Geospatial Consortium (OGC) and the ISO TC211" [Grö+12]. The model was initially developed by the Special Interest Group 3D (SIG3D) in 2002 and later became an OGC standard in 2008. In 2012, CityGML version 2.0.0 was released.

Most of common city objects and features, such as buildings, vegetation, water, terrain, traffic, tunnels, bridges, etc. can be described in CityGML. Unlike other models, CityGML is capable of including not only 3D geometry and graphical appearances, but also 3D topology and semantic properties of city objects [KGP05]. Moreover, CityGML can represent city objects and features in five different Levels of Details (LOD), namely from level 0 to 4, as illustrated in Figure 2.2.

To support city objects from different thematic areas, the CityGML data model is decomposed into a core module and other respective thematic extension modules, namely `Appearance`, `Bridge`, `Building`, `CityFurniture`, `CityObjectGroup`, `Generics`, `LandUse`, `Relief`, `Transportation`, `Tunnel`, `Vegetation`, `WaterBody` and `TexturedSurface` as shown in Figure 2.3. In the context of this thesis, only the module `Building` (with `Appearances`) is considered. However, the implementation for other modules can in principle be done in a similar manner.

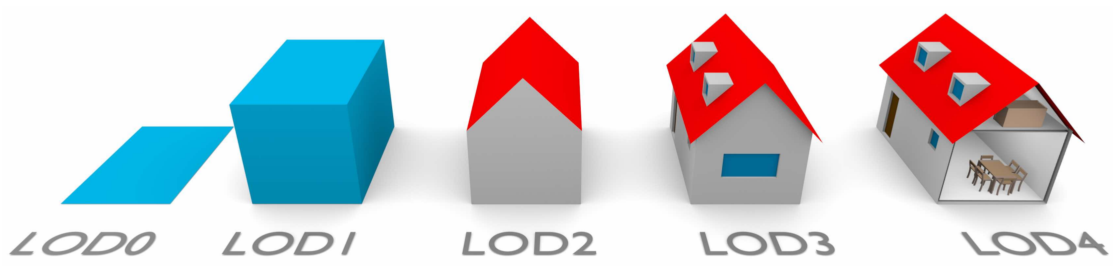


Figure 2.2: Five different Levels of Details (LOD) 0 - 4 of a building representation in CityGML. Source: Delft University of Technology (TU Delft).

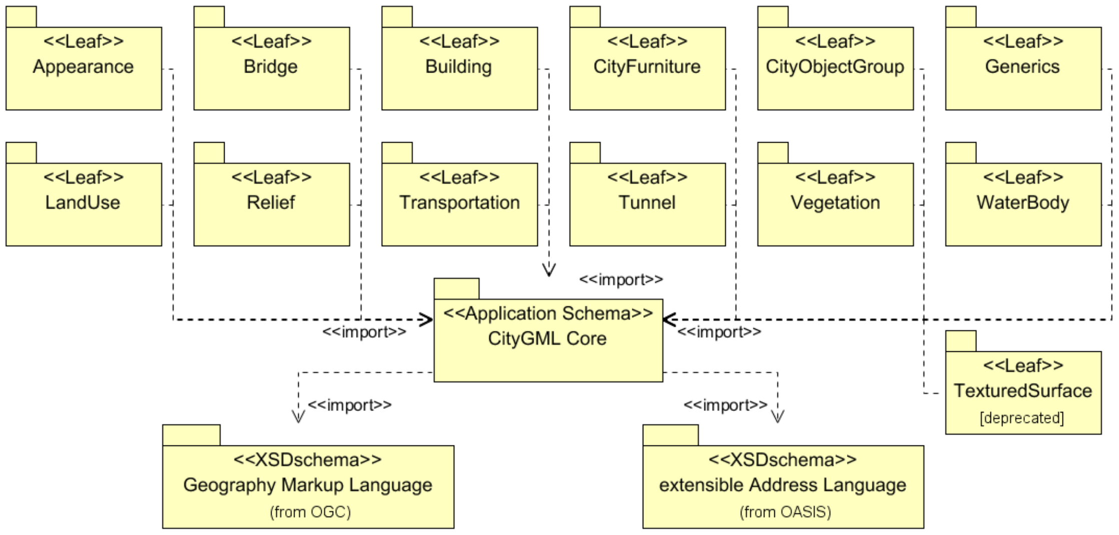
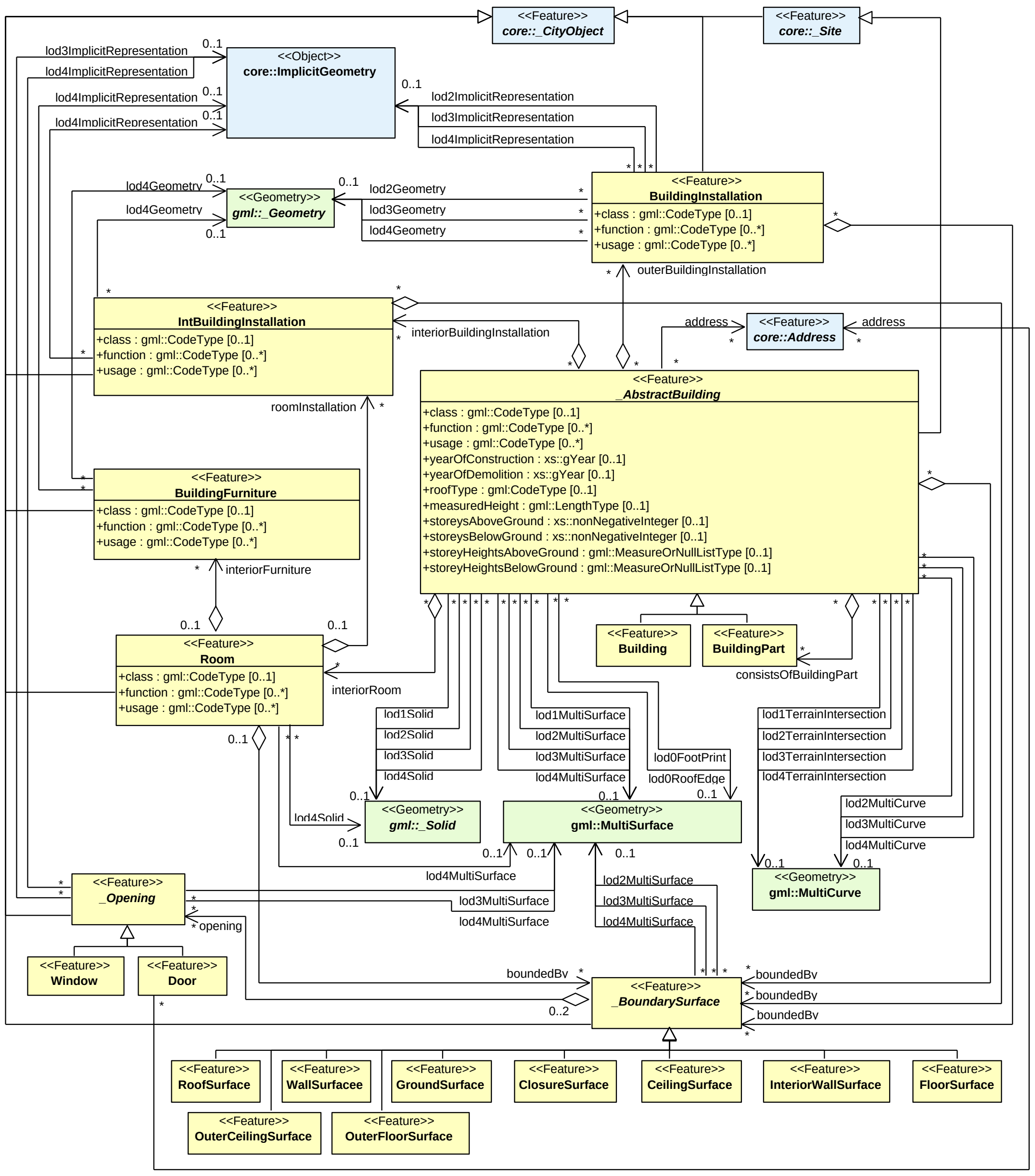


Figure 2.3: UML package diagram illustrating the separate modules of CityGML and their schema dependencies. Each extension module (indicated by the leaf packages) further imports the GML 3.1.1 schema definition in order to represent spatial properties of its thematic classes. For readability reasons, the corresponding dependencies have been omitted. Caption and figure taken from [Grö+12].

The `Building` module enables "the representation of thematic and spatial aspects of buildings, building parts, building installations, and interior building structures in five levels of detail (LOD 0 – 4)" [Grö+12]. Objects that belong to this module are defined in a class hierarchy illustrated in Figure 2.4.



<<Feature>> core::_CityObject
<<Feature>> core::_Site
<<Object>> core::ImplicitGeometry
<<Geometry>> gml::_Geometry
<<Feature>> BuildingInstallation
+class : gml::CodeType [0..1]
+function : gml::CodeType [0..*]
+usage : gml::CodeType [0..*]
<<Feature>> IntBuildingInstallation
+class : gml::CodeType [0..1]
+function : gml::CodeType [0..*]
+usage : gml::CodeType [0..*]
<<Feature>> core::Address
<<Feature>> _AbstractBuilding
+class : gml::CodeType [0..1]
+function : gml::CodeType [0..*]
+usage : gml::CodeType [0..*]
+yearOfConstruction : xs::gYear [0..1]
+yearOfDemolition : xs::gYear [0..1]
+roofType : gml:CodeType [0..1]
+measuredHeight : gml::LengthType [0..1]
+storeysAboveGround : xs::nonNegativeInteger [0..1]
+storeysBelowGround : xs::nonNegativeInteger [0..1]
+storeyHeightsAboveGround : gml::MeasureOrNullListType [0..1]
+storeyHeightsBelowGround : gml::MeasureOrNullListType [0..1]
<<Feature>> BuildingFurniture
+class : gml::CodeType [0..1]
+function : gml::CodeType [0..*]
+usage : gml::CodeType [0..*]
<<Feature>> Room
+class : gml::CodeType [0..1]
+function : gml::CodeType [0..*]
+usage : gml::CodeType [0..*]
<<Feature>> Building
<<Feature>> BuildingPart
<<Geometry>> gml::_Solid
<<Geometry>> gml::MultiSurface
<<Geometry>> gml::MultiCurve
<<Feature>> _Opening
<<Feature>> Window
<<Feature>> Door
<<Feature>> _BoundarySurface
<<Feature>> RoofSurface
<<Feature>> WallSurfacee
<<Feature>> GroundSurface
<<Feature>> ClosureSurface
<<Feature>> CeilingSurface
<<Feature>> InteriorWallSurface
<<Feature>> FloorSurface
<<Feature>> OuterCeilingSurface
<<Feature>> OuterFloorSurface
lod3ImplicitRepresentation
lod4ImplicitRepresentation
lod2ImplicitRepresentation
lod2Geometry
lod3Geometry
lod4Geometry
outerBuildingInstallation
interiorBuildingInstallation
address
roomInstallation
interiorFurniture
interiorRoom
consistsOfBuildingPart
lod1Solid
lod2Solid
lod3Solid
lod4Solid
lod1MultiSurface
lod2MultiSurface
lod3MultiSurface
lod4MultiSurface
lod0FootPrint
lod0RoofEdge
lod1TerrainIntersection
lod2TerrainIntersection
lod3TerrainIntersection
lod4TerrainIntersection
lod2MultiCurve
lod3MultiCurve
lod4MultiCurve
opening
boundedBy
0..1
0..2
*

Figure 2.4: UML diagram of CityGML's building model. Prefixes are used to indicate XML namespaces associated with model elements. Element names without a prefix are defined within the CityGML `Building` module. Caption and figure taken from [Grö+12].

2.2 XML Processing

2.2.1 XML Parsing

Since CityGML is an application schema of GML, which is an extension of XML, extracting and processing data stored in its documents can theoretically be done by XML parsers. Depending on how such parsers handle the data structure, they are divided into two categories: model-based and stream-based (or event-based) [SN09], most of which are available as (Java) APIs.

Model-based APIs

One of the most well-known representatives of the model-based approach is the cross-platform, language-independent **Document Object Model (DOM)**, which reads the entire input document into main memory and treats it as a tree structure, whose nodes represent data elements stored in the input document [Kes+15] (see Figure 2.5). This means that the model has full control over object navigation and data manipulation (*pull-parsing*), such as programmatically finding objects by names or inserting new as well as deleting existing elements in the document tree. Thus, the DOM is very popular in most modern web browsers as it provides a fast and powerful way to handle web documents. The major drawback is however its inefficient main memory consumption while parsing large XML documents.

Stream-based APIs

The **Simple API for XML (SAX)**, one of the stream-based APIs, is basically the polar opposite of DOM. Namely, while DOM loads documents as a whole, a SAX parser sequentially processes only a piece of input data at a time. Its workflow is based on events while streaming, such as when a start or end-tag of a specific element is encountered. When triggered, these events report to the parser (callback) so that actions can be taken accordingly (*push-parsing*). Afterwards, the parser discards irrelevant information and loads a new piece of data into main memory, and so on until the whole document is processed. Thus, the SAX requires much less memory compared to the DOM. This is a strength while processing large XML datasets but also a disadvantage in some other certain scenarios, where access to the entire document is required, such as validating or transforming XML documents using XSLT and XPath, in which DOM is a better approach. Moreover, the SAX cannot re-read elements that it has discarded from previous iterations. Finally, the SAX is a read-only approach. Therefore, whether the DOM or SAX is suitable depends on how the application processes XML documents.

```
<html>
   <head>
      <title>Some Title</title>
   </head>
   <body>
      <h1>First Heading</h1>
      <p>First paragraph.</p>
   </body>
</html>
```

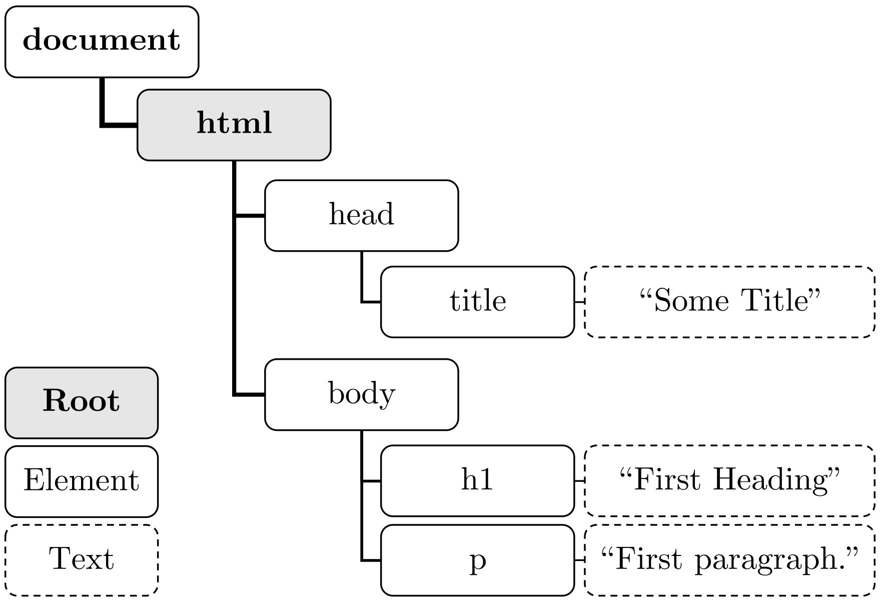


Figure 2.5: An example of a web document and its DOM representation.

Unlike the DOM, no formal specification for SAX exists. Hence, its Java implementation is considered to be a standard.[1]

Another alternative is the **Streaming API for XML (StAX)**, which is a combination of both the DOM and SAX. Technically, the StAX is also a stream-based API like SAX and thus applicable to large XML documents, but similarly to DOM, it only pulls information from the parser when needed (*pull-parsing*). Additionally, StAX can both read and write XML documents.

[1]`http://www.saxproject.org/`.

2.2.2 XML Data Binding

XML data binding is a concept that represents XML documents as a single or a set of business objects, which are made of instance attributes and associations with other business objects. This allows access to data stored in XML documents via the business objects without having to rely solely on the DOM or SAX.

To achieve this, an XML data binder converts an XML document into a business object with the help of its associated XML schema. This process is called unmarshalling [Ora17]. The reverse process is called marshalling, where the object is converted back to XML. Since the business objects and its contents are completely held in main memory, the XML data binding is considered a model-based approach [SN09]. However, by combining JAXB with SAX or StAX, it is possible to divide XML documents into smaller chunks, each of which can sequentially be mapped and processed in main memory (partial unmarshalling and marshalling).

XML data binders are a suitable approach to complex and changing XML schemata like CityGML. The **Java Architecture for XML Binding (JAXB)** is one of such XML data binders capable of unmarshalling XML to Java objects in main memory and vice versa (see Figure 2.6). The advantage of JAXB over DOM and SAX is that users are not required to possess a thorough understanding of XML data structure. Moreover, mapped instance objects are organized in a class hierarchy corresponding to the given XML schema. This offers developers more freedom to focus on the application domain rather than the internal XML parsing mechanism.

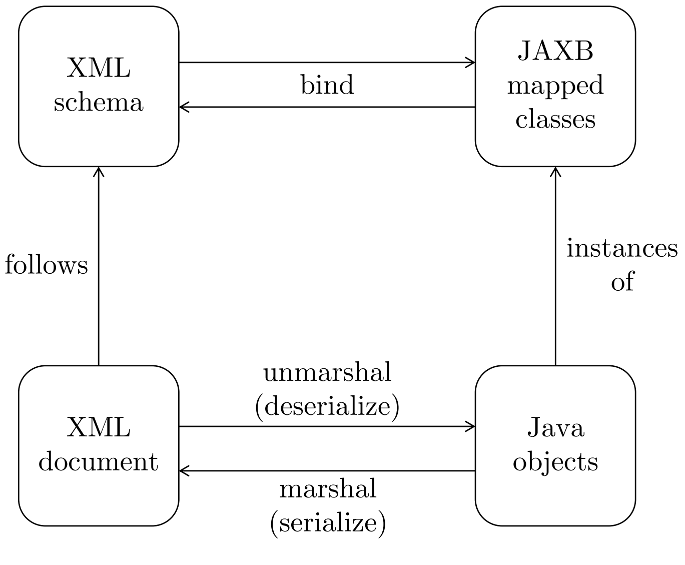


Figure 2.6: The JAXB binding process. Adapted from [Ora15].

2.3 citygml4j - The Open Source Java API for CityGML

Since CityGML is XML-based, reading, processing, writing CityGML datasets and developing CityGML-aware software often require a thorough understanding of the underlying data structure. However, not all software developers are familiar with XML technology. Therefore, to enable and ease such CityGML related work, an open source Java class library and API named "citygml4j" was developed [Nag17].

The citygml4j library makes use of the JAXB for binding and processing XML documents as explained previously in Section 2.2.2. At its core, based on the XML schema definitions of CityGML, citygml4j unmarshals (or deserializes) CityGML instance documents accordingly into corresponding Java objects, whose contents and structures are stored and organized in tree representations of given instance documents (see Figure 2.6). Java developers can then manipulate these produced objects for their own software developments without having to acquire knowledge of parsing XML data structure beforehand. Furthermore, citygml4j is also capable of marshalling (or serializing) Java objects back into CityGML elements.

Some key features of citygml4j listed by [3DC17] are:

- Full support for CityGML version 2.0.0 and 1.0.0 (read-only support for version 0.4.0),
- Support for CityGML specific subset of GML 3.1.1,
- Support for the eXtensible Address Language (xAL),
- Support for user-defined CityGML Application Domain Extensions (ADE).

Like CityGML, citygml4j is part of 3D City Database software and can be used free of charge [3DC17].

2.4 Graph Database in Neo4j

This section introduces the concept of a graph database and its application for comparing massive CityGML datasets. For this purpose, the software Neo4j is employed. A brief introduction of its graph data model and database transaction mechanism follows.

2.4.1 From Relational to Graph Database

Relational Database

First described in 1969, the relational model has become one of the foundation concepts for storing and maintaining data [Cod70]. The model organizes data into a set of tables (or "relations"), each of which consists of columns and rows (or "tuples"). Every row in a table is identified by their unique (or primary) key. Rows between tables are linked by foreign keys, which can be represented as additional columns holding the unique keys of referenced rows (see Figure 2.7). A database, whose organization is based on the concepts of the relational model, is called a relational database. Software systems employed to maintain such databases are called Relational Database Management Systems (RDBMS). The Structured Query Language (SQL) is used in most RDBMS.

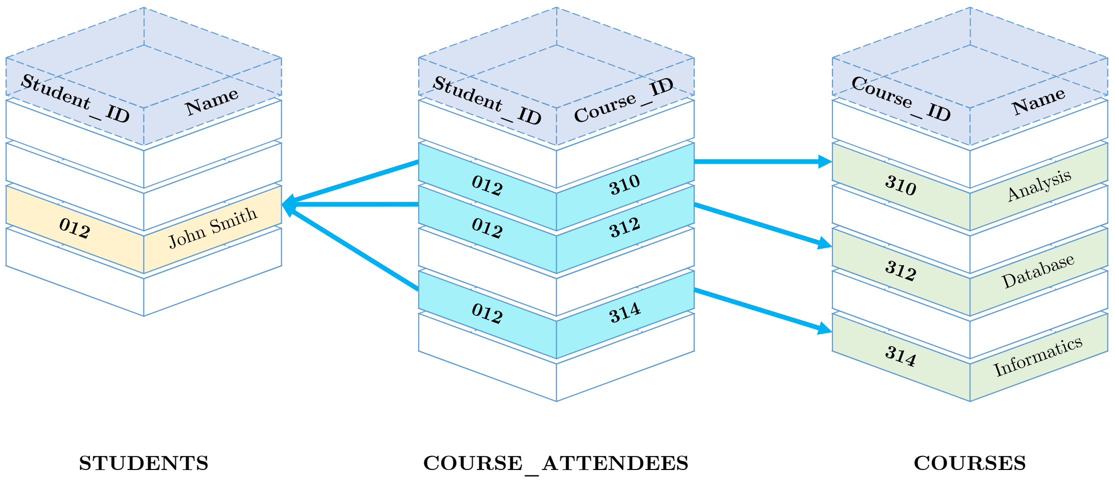


Figure 2.7: An example of the relational model. Each row of table `STUDENTS` (left) is uniquely identified by a `Student_ID` key. Similarly, a course is identified by a unique `Course_ID` in `COURSES` (right). Student and course entries are linked together by a series of rows in `COURSE_ATTENDEES`, whose columns contain foreign keys of respective referenced tables. Adapted from [Neo17a].

A sustainable relational database often requires strictly designed model structures with a predetermined number and appointed types of columns in each table. Moreover, as illustrated in Figure 2.7, references to rows between tables hold values of primary keys from each respective table and therefore require these keys to be non-empty (e.g. with constraints). In other words, the foreign keys stored in table `COURSE_ATTENDEES` must be available in both tables `STUDENT_ID` and `COURSE_ID`. Then, the `JOIN` operator between `STUDENTS` and `COURSE_ATTENDEES` basically looks for and match primary with foreign keys stored in rows from relevant tables. If many-to-many relationships exist, an associative (or junction) table containing all referenced foreign keys is needed. This process costs computational and memory resources and causes the query response time to grow exponentially with respect to table sizes.

Graph Database

In many use case scenarios, accessing only a fraction of a database and its related data at a time has much higher priority than iterating over the entire set of tables (e.g. to find boundary surfaces of a given building in a city). In this context, graph database is an appealing alternative to its relational counterpart, especially when data contains hierarchical information and complex associations.

Graph databases make extensive use of graph entities such as properties, nodes and edges (or relationships). Data are mainly stored in nodes and their properties, while the relationships between data items are represented by edges between nodes. This way, references are explicitly expressed in a graph database without having to rely on additional auxiliary foreign keys as in the relational model. Since graph entities are connected directly together, a query on their relationships is generally much simpler than in a conventional relational database because `JOIN` operators are no longer required (see Figure 2.8).

However, graph and relational databases do not replace each other. On the contrary, they both complement each other due to their different types of application domains. Relational databases are well-suited to flat data structures containing a large number of records, where the maximum depth of relationships between entities is low (i.e. level one or two). On the other hand, graph databases are specifically useful in handling relationships of a given data model, whose elements are connected by a deep network of complex references.

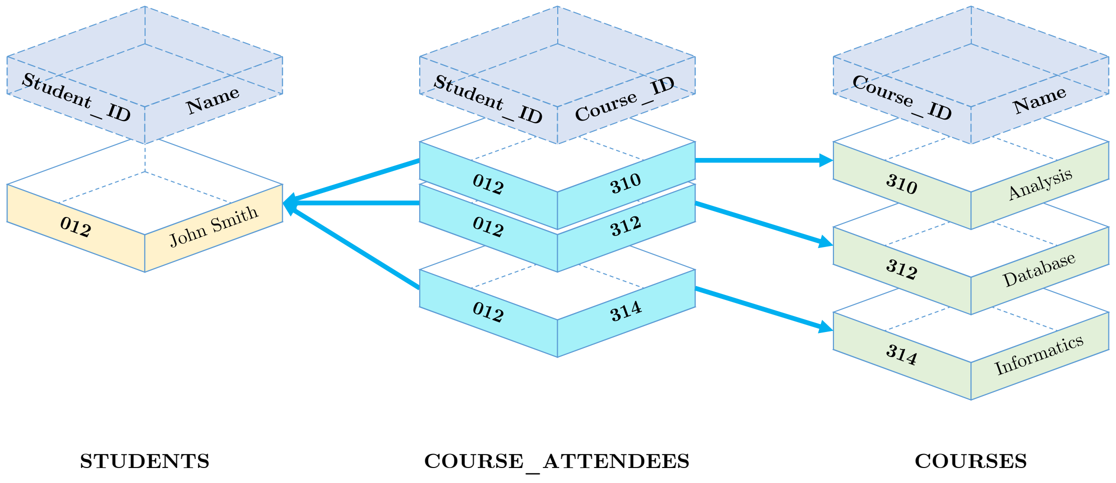


Figure 2.8: Relevant connected data items from Figure 2.7. `JOIN` operators are no longer required. Adapted from [Neo17a].

2.4.2 CityGML in Graph Database

CityGML datasets can be stored and processed in a graph database. Some of the main reasons are:

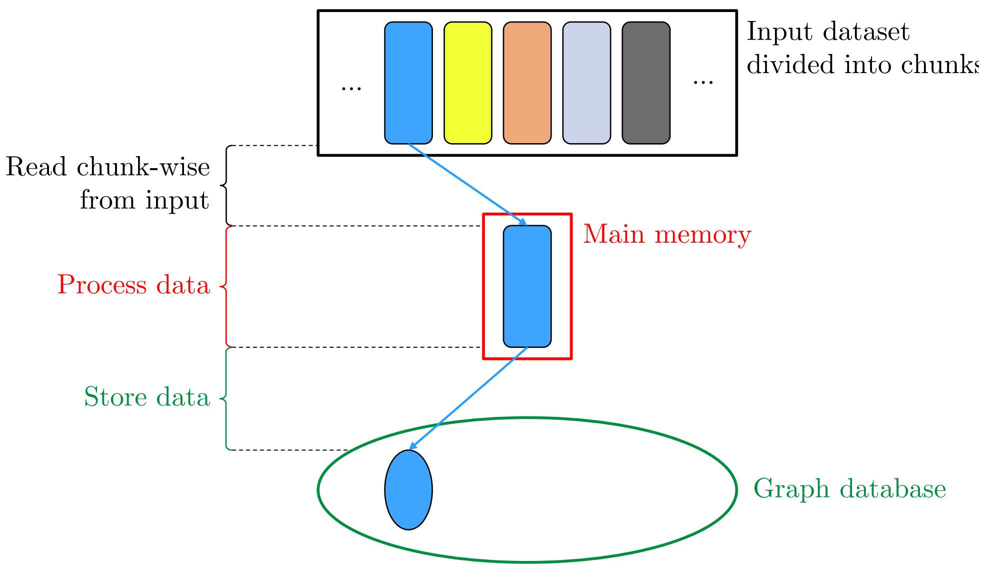
- CityGML elements belong to a complex hierarchical structure containing multi-level deep associations. Moreover, a CityGML element can be pointed to by multiple parents forming a cycle. Thus, CityGML is basically considered a graph data structure;
- Graph databases allow fast access to a data entity and its related data (e.g. a building and its boundary surfaces);
- Syntactic ambiguities in XML such as between in-line and hyper-link (XLink) declarations can be solved in graphs;
- CityGML datasets can be very big in size (e.g. approximately up to 20 GB in the test use case scenario of the city of Berlin). Hence, reading such huge datasets completely into main memory should be the last resort. Therefore, an approach that reads large CityGML in chunks (piece by piece) is employed, where a graph database is needed to store the processed information for later use before the main memory flushes and replaces the old chunk with the next one (see Figure 2.9).

Then, the comparison between two CityGML datasets is performed by matching their graph representations stored in the graph database.

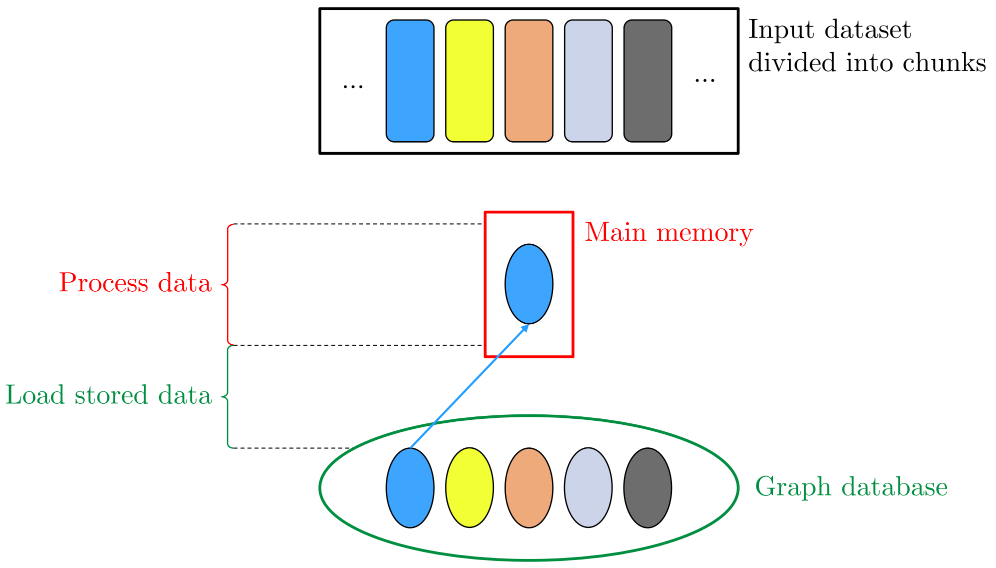
2.4.3 Neo4j Graph Database Management System

In order to create, manipulate and maintain massive graphs, the Neo4j graph database is employed. Neo4j is a graph database management system developed by Neo Technology, Inc. As of February 2017, Neo4j is the world's most popular graph database according to [DBE17]. Neo4j is a native, high performance graph database built specifically for storing and processing graphs. It takes advantage of connections between data stored in nodes and edges, thus accelerates query speed by ignoring data that are not connected to relevant nodes. It is also a fully ACID (Atomicity, Consistency, Isolation, Durability) transactional database.

Neo4j is available in three editions: Community, Enterprise and Government. Both Enterprise and Government Editions are commercial and offer a number of comprehensive functionalities such as clustering, hot backups, advanced monitoring and more. The Community Edition is free of charge but lacks these features. In the context of this thesis, the Community Edition is employed in the implementation.



(a) Large input CityGML dataset is divided into small pieces (rectangles), which are successively loaded into main memory (red). There, the data pieces are processed sequentially (here: blue rectangle first). The processed information is then stored in a graph database (green) for later use. The main memory then flushes and loads new piece of data (yellow rectangle).



(b) All input pieces that are previously processed and stored in the graph database (ellipses in colours based on their origin) can now be loaded back to main memory.

Figure 2.9: An illustration of reading large CityGML datasets piece by piece. The main memory serves as a "buffer" while the graph database is a storage for processed information.

2.4.4 Graph Structures in Neo4j

Neo4j stores graph data items in properties, nodes and relationships, where:

Property Flat data such as texts, numbers, booleans, etc. can be stored in properties. Every property must exist inside a node or a relationship, where they are uniquely identified by their respective names. In other words, no two properties have the same name in a node or relationship. Properties in graphs correspond to column values in an equivalent relational database.

Node Nodes are central data entities in Neo4j. A node can contain an arbitrary number of labels and properties. Labelled nodes are indexed and can be retrieved using one of their assigned labels (schema indexing). Nodes are linked together by relationships. Graph nodes correspond to rows in entity tables from an equivalent relational database.

Relationship Relationships connect nodes together. In Neo4j, relationships are directed and thus point from a start to an end node. However, relationships can be traversed in both directions (namely in `INCOMING` and `OUTGOING` direction). Each relationship has a relationship type and like nodes, can contain an arbitrary number of properties. Graph relationships correspond to foreign keys in an equivalent relational database.

An illustration of properties, nodes and relationships in Neo4j can be found in Figure 2.10.

Furthermore, for performance and maintenance purposes, Neo4j (version 2.0 and later) introduces an optional schema for graphs. Schema consists of indices and constraints. Namely:

Index To enable efficient querying on data, Neo4j creates a redundant copy of graph entities and stores it in database storage. This copy is called index. An index can be created automatically for properties of all nodes of the same label (schema indexing) or manually for nodes of different labels (legacy indexing). The drawbacks of indices are the additional required storage and slower disk reads and writes. Therefore, indices should only be used in scenarios, where the number of queries requested on a given property far exceeds that of modification operations.

Constraint Another aspect of graph schema is constraints on nodes and relationships. Neo4j allows unique and existence constraints on node properties as well as existence constraints on relationship properties. In case of unique constraints, an index is implicitly created for affected data.

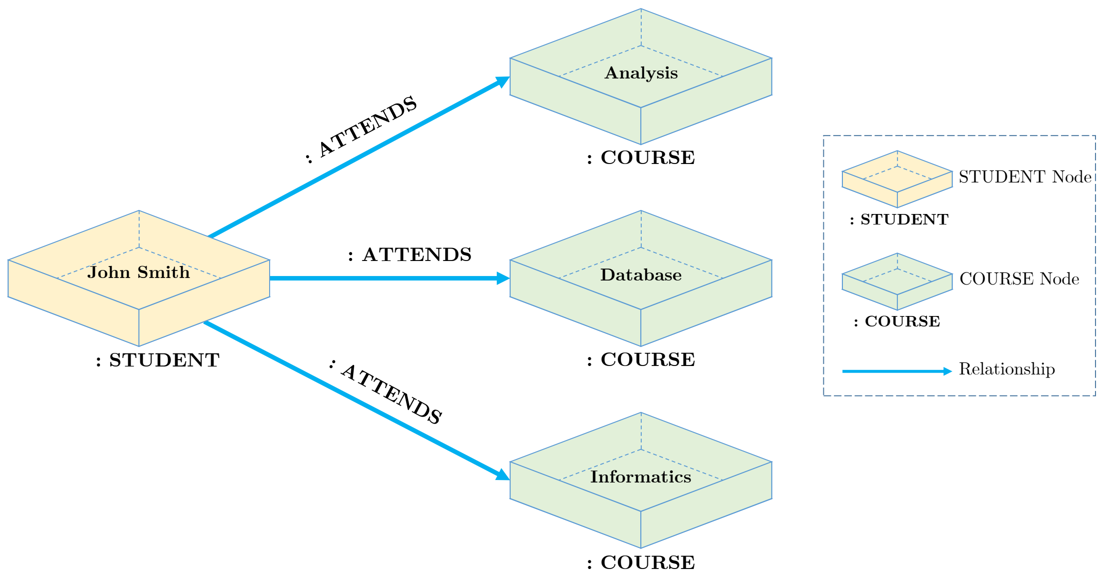


Figure 2.10: An example of a graph representation of the data model shown in Figure 2.8 in Neo4j. `STUDENT` and `COURSE` objects are stored as nodes, whose labels are `:STUDENT` and `:COURSE` respectively. Name values of students and courses are held in properties of respective nodes. Relationships connect students and courses together, whose start node is a `STUDENT` and end node is a `COURSE`. These relationships have the same relationship type, namely `:ATTENDS`. Adapted from [Neo17a].

2.4.5 Developing in Neo4j

Two important aspects of interacting with Neo4j are querying and manipulating database. For these purposes, Neo4j provides its own declarative query language called Cypher and the Bolt protocol. For the more comprehensive use of database, the Java Core API is employed.

Cypher and Bolt Protocol

Cypher is a declarative query language developed for Neo4j. Despite being relatively simple, Cypher is well capable of expressing very complicated queries. Since Cypher is designed to be understandable to humans, it is mostly aimed at operations professionals.

The structure of Cypher is basically comparable to that of SQL in relational model.

However, different key words, such as `MATCH`, `WHERE` and `RETURN` are used to form queries. For example, Listing 2.1 describes a query that returns a building with a given ID from the database.

Listing 2.1: A simple Cypher query to retrieve building(s) with ID equal to “Some_ID”.

```
MATCH (b:BUILDING)
WHERE (b.id = ‘Some_ID’)
RETURN b
```

Cypher queries can be executed directly from the Neo4j browser-based client or via Neo4j’s official drivers. Neo4j drivers enable application access to graph database and are available in .NET, Java, JavaScript and Python. To establish communication to database regardless of used drivers and database versions, the Bolt protocol is applied. The Bolt protocol accepts Cypher queries, executes them and returns results that can be further processed in the supported programming languages of the respective driver. For example, the previously shown Cypher query can be executed in Java driver with the code excerpt shown in Listing 2.2.

Listing 2.2: An example of Neo4j Bolt in Java.

```
Driver driver = GraphDatabase.driver("bolt://localhost:7687",
                      AuthTokens.basic("neo4j", "neo4j"));

try (Session session = driver.session()){
        try (Transaction tx = session.beginTransaction()) {
                StatementResult result = tx.run(
                        "MATCH (b:BUILDING) " +
                        "WHERE (b.id = {parameterId}) " +
                        "RETURN b",
                        parameters("parameterId", "Some_ID"));

                while (result.hasNext()) {
                        Record record = result.next();
                        System.out.println(
                                "GMLID: " + record.get("id").asString())
                        );
                }
        }
}

driver.close();
```

Compared to other services, the Bolt protocol is relatively young and currently in active development. However, due to its user friendliness and platform independence across multiple driver and database versions, Bolt is a promising alternative aimed at professionals and developers outside of the Java-ecosystem. For more information, please refer to the Neo4j Developer Manual [Neo17b].

Neo4j Java Core API

In contrast to the Bolt protocol being designed specifically for multiple drivers and programming languages, the Java Core API is aimed at Java developers. Since Neo4j is developed in Java, the Neo4j Java Core API (or Neo4j Java Embedded) exists as one of the most powerful interfaces in Neo4j. It allows comprehensive and fully customizable database operations (such as creating, updating and deleting database), as well as querying and traversing data in database. An example illustrating some of these operations in Java is shown in Listing 2.3.

Listing 2.3: An example of Neo4j Java Core API.

```java
// Create and start a new graph database
GraphDatabaseService graphDb
        = new GraphDatabaseFactory().newEmbeddedDatabase("DB_PATH");
// Always wrap database operations in transactions
try (Transaction tx = graphDb.beginTransaction()) {
        // Create a node representing a building object
        Node building = graphDb.createNode();
        // Assign an ID property to this building
        building.setProperty("id", "Building_ID");
        // Create a node representing a ground surface
        Node groundSurface = graphDb.createNode();
        // Assign an ID property to this ground surface
        ground.setProperty("id", "GroundSurface_ID");
        // Create a relationship between these two nodes
        building.createRelationshipTo(groundSurface,
                                        CityGMLRelTypes.BOUNDED_BY_SURFACE);
        // Release database resources from current transaction
        tx.success();
}
// Close the database
graphDb.shutdown();
```

To ensure the ACID properties in Neo4j, all database operations, regardless of whether being executed via Bolt or in Java Core API, must be wrapped in transactions (see Line 5 in Listing 2.2 and Line 5 in Listing 2.3). This is a "conscious design decision" [Neo17c]. If an attempt is made to access the database outside of a transaction, a `NotInTransactionException` will be thrown. After a transaction is started, locks are acquired and assigned accordingly to affected database objects. These objects remained locked and held in main memory as long as the transaction is still active. To release locks and affected objects, the transaction must be marked as either a success or failure. In case of a successful transaction (Line 18 of Listing 2.3), changes are committed to the database. In case of a failure, to maintain data integrity, all operations wrapped in the transaction are rolled back and nothing is committed (see Figure 2.11). Finally, in both cases, the transaction will then release acquired locks and affected objects from main memory. For more information on the transaction management, please refer to the Neo4j Java Developer Reference [Neo17c].

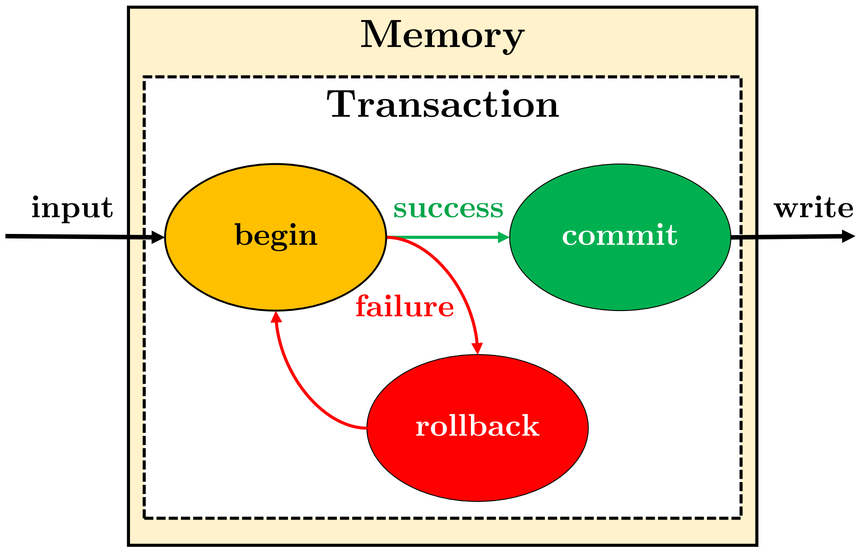


Figure 2.11: An example of the transaction management in Neo4j. All database operations must be wrapped in transactions. After a transaction is started (orange), affected nodes and relationships are locked and held in main memory. Only after the transaction is marked as either "success" or "failure" can these locks and data items be released from main memory. If the transaction is successful, changes will be committed to database (green). However if the transaction failed, all uncommitted changes will be rolled back (red) to the initial state.

One of the most important advantages provided by the Java Core API over other services is the full control of how nodes, relationships and their properties are built (Lines 7, 9, 11, 13 and 15 of Listing 2.3). Moreover, since nodes and relationships are stored as Java objects, they are fully applicable to the object-oriented concepts. In addition, the fact that each node and relationship can easily be reused and referenced makes the entire workflow more intuitive and manageable (Lines 9, 13 and 15). Furthermore, since almost all other services (e.g. Cypher queries, the Bolt Protocol, the Traversal Framework, etc.) are included in the Java Core API, the Java Core API is generally a more universal approach to a broader range of applications.

Considering the above-mentioned advantages, the Neo4j Java Core API is extensively employed throughout the implementation process of this thesis.

2.5 R-tree Data Structure and Neo4j Spatial

2.5.1 R-tree Data Structure

R-trees are tree data structures developed especially for spatial indexing, i.e. storing and retrieving geographic information, such as locations of rectangles and polygons. The "R" in R-tree stands for "rectangle". The R-tree was introduced in 1984 [Gut84] and not only has it since been recognized as one the most well-known tools used in both theoretical and applied areas, it is also a "necessary" approach to enabling fast handling of multi-dimensional data in spatial databases and geographical information systems [Man+05]. The main concept of R-trees is that geometric objects spatially located near to each other can be grouped into a larger object containing their minimum bounding box (or rectangle). Each of these objects is represented as a leaf in the tree, while the aggregated object containing the minimum bounding box is assigned to the next higher level (see Figures 2.12 and 2.13). Recursively, multiple neighbouring internal nodes can be grouped again to form a higher node on the next level. This means that if a query geometry does not intersect a bounding box, then it also cannot reach any of the contained objects. As a result, like in most tree data structures, spatial queries, such as intersection and nearest neighbour search, are very efficient, as most irrelevant nodes can be avoided. Moreover, since R-trees are also balanced search trees, the query response time is generally bound by the depth of leaf nodes. Namely, the average time complexity of a search operation in R-trees is

$$\mathcal{O}(\log_M n),$$

where M is the maximum number of entries contained in an internal (or non-leaf) node and n is the total number of stored nodes. Note that since R-tree rectangles can overlap, multiple paths down to the leaves might be required to be searched.

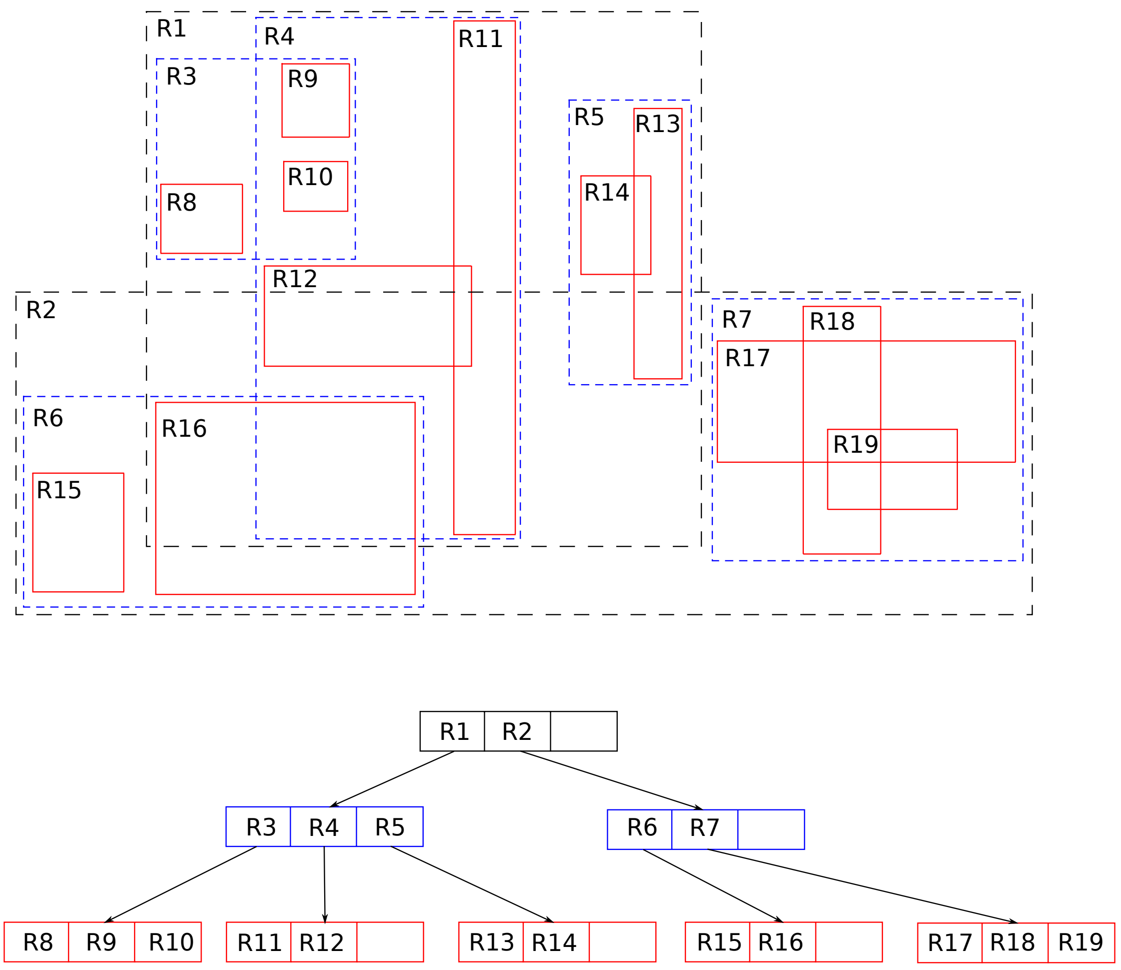


Figure 2.12: An example of an R-tree. Leaf nodes are represented as red rectangles, while their minimum bounding rectangles are shown in blue and grey. Source: [Bac10].

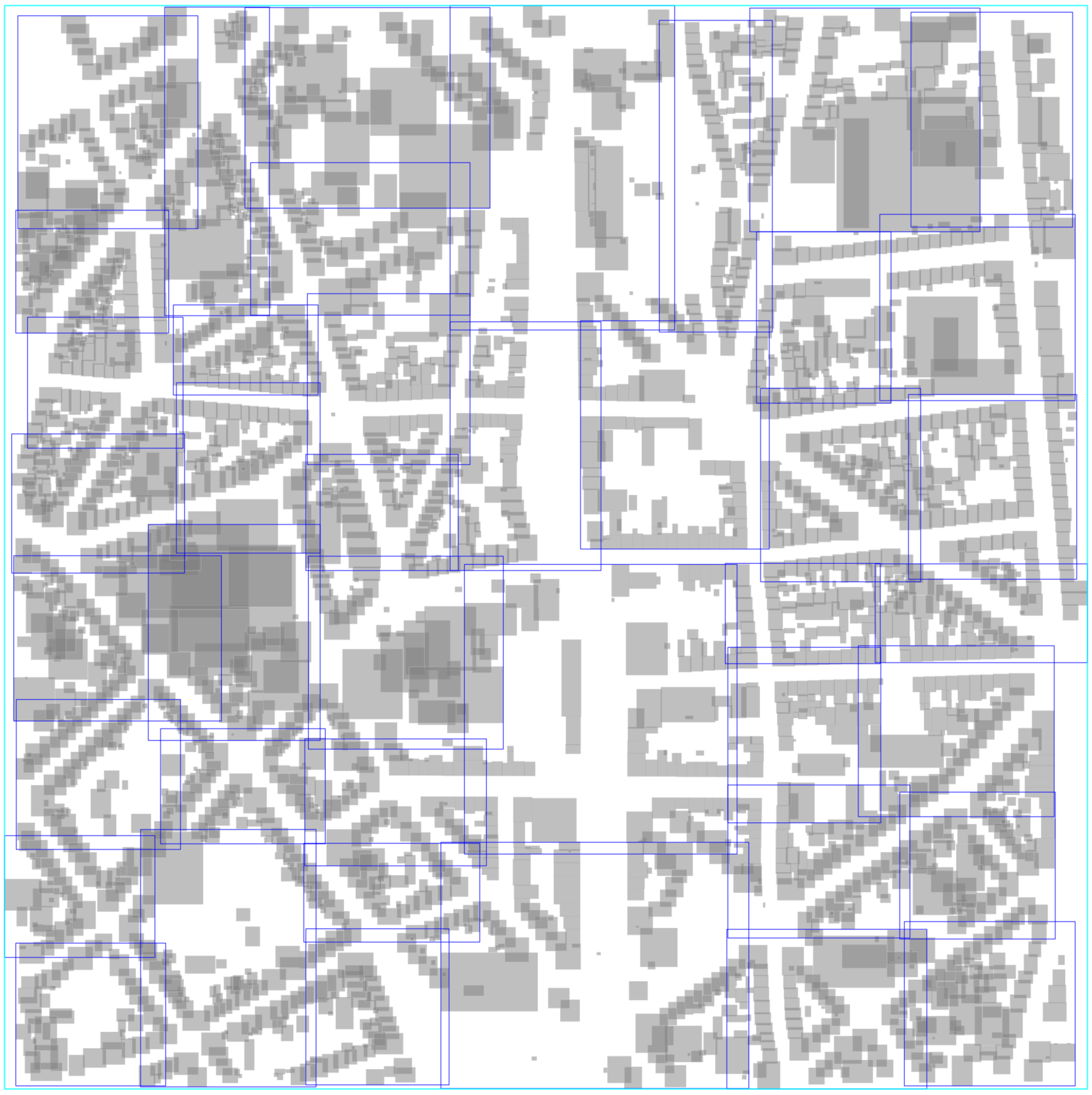


Figure 2.13: An R-tree representation of an excerpt from the CityGML dataset provided by the North Rhine-Westphalia state, Germany. The R-tree image was produced automatically by Neo4j Spatial during the mapping process.

2.5.2 Neo4j Spatial

The Neo4j Spatial[2] is an open-source utility plug-in for Neo4j. Neo4j Spatial extends the base Neo4j version with a number of key features, such as utilities for importing ESRI Shapefile or Open Street Map files, support for all common geometry types in two-dimensional space (e.g. `Point`, `LineString`, `Polygon`, . . .), etc. but most importantly, it allows spatial operations on data by enabling an R-tree spatial index to already stored graph entities in Neo4j. This means that:

- Search operations on stored geographical data should generally be executed in logarithmic time complexity as an R-tree is in use (as explained previously in Section 2.5.1);
- Spatial index can be applied to already available data in Neo4j without interfering with how they are originally stored.

The latter can be achieved by providing an "adapter" that connects data to their respective geometry (see Figure 2.14). Geometric objects are handled internally by the Java Topology Suite (JTS) contained in this plug-in. Available spatial queries applicable in two-dimensional space are listed as follows [Neo17e]:

- Contain
- Cover
- Covered by
- Cross
- Disjoint
- Intersect
- Intersect Window
- Overlap
- Touch
- Within
- Within Distance

In Neo4j Spatial, spatial indices are organized in layers, which are composed of nodes stored internally in the same Neo4j database as that of referenced data. Note that the R-tree's internal nodes (with the exception of the root node) are unlabelled and hence excluded from schema indexing. Once added, the geometry objects are stored in a balanced R-tree of respective layer. Then, based on the type of requested spatial queries (as listed above), the R-tree is traversed and a geometry with all its referenced data are returned with the help of the constructed adapter.

[2] `https://github.com/neo4j-contrib/spatial`

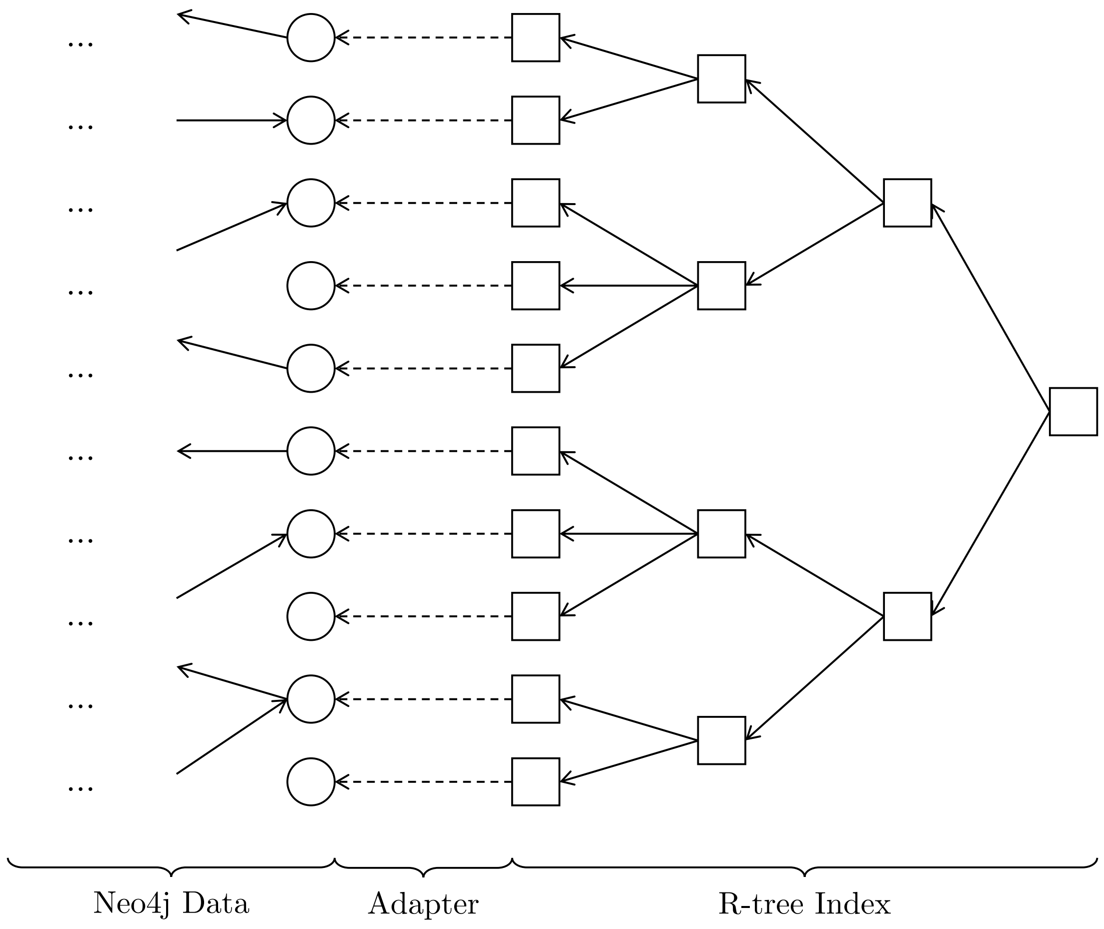


Figure 2.14: Illustration of an adapter (middle) connecting spatial indices in Neo4j Spatial (right) with data already stored in Neo4j (left).

2.6 Web Feature Service (WFS)

Large geospatial databases, such as the CityGML dataset of Berlin in the test use case, are often stored in a central server and managed via web services. The Open Geospatial Consortium (OGC) Web Feature Service (WFS) is one of such services. Applications that operate on WFS-enabled data sources should thus also be WFS-compatible.

The WFS is a platform independent interface standard that enables the creation, modification and exchange of geographic features across the web. One of its major advantages over other protocols is the ability to retrieve and modify requested data at the feature and feature property level with "fine-grained precision" [Vre14]. Instead of unnecessarily providing a whole data file that contains the requested information, the WFS responds with only features that met the criteria issued by querying clients. The clients can then modify or perform a spatial analysis on the received data based on their needs.

2.6.1 WFS Communications

In general, as illustrated in Figure 2.15, the WFS serves as a communicating component located between the geospatial database and the clients. To enable transporting of geographic features, the data held in the geospatial database must conform to the GML schema (e.g. CityGML as a GML application schema in particular). In addition, however, other formats like ESRI shapefiles are also allowed. In the following sections, the WFS requests and responses between the clients and database are briefly introduced.

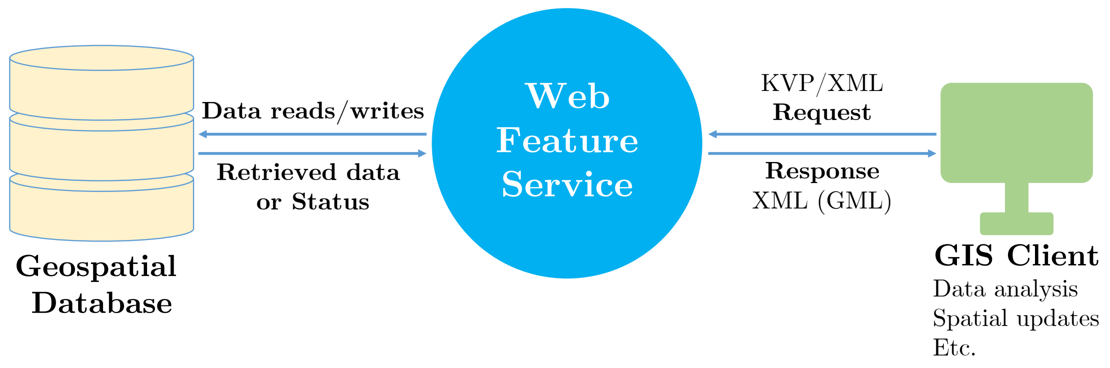


Figure 2.15: An overview of the Web Feature Service (WFS) in interaction with GIS clients and geospatial database.

Client Request

Multiple clients can simultaneously send requests to the WFS, which then executes them accordingly (the maximum number of clients and requests that can be processed simultaneously depends on the WFS server configurations). A WFS request can be declared in two types of encodings: Key/Value Pair (KVP) and XML.

KVP The KVP is encoded in HTTP GET requests as query strings. An example of an HTTP GET request with its encoded KVP of two keys `key1`, `key2` and their respective values `value1`, `value2` is as follows:

```
http://example.com/web/service&key1=value1&key2=value2
```

The advantages of KVP are its compactness and simplicity. It suffices for most read-only queries. However, requests that are encoded in KVP are often not expressive enough for more complicated scenarios, such as updating a feature with a new content.

XML HTTP POST (and SOAP) request contents are encoded in XML. The advantage of this encoding is its strong expressiveness and can be employed in all scenarios. In other words, HTTP GET requests can be replaced by corresponding HTTP POST ones. The drawback is that HTTP POST contents are too complex to fit in a URL and thus must be passed as additional parameters. Listing 2.4 illustrates a simplified example of an HTTP POST content equivalent to the previous HTTP GET request.

Listing 2.4: An example of an HTTP POST request.

```
POST
Host: http://example.com/web/service

HEADERS
Content-Type: application/xml

PAYLOAD
<?xml version="1.0" encoding="UTF-8"?>
<OperationType service="ServiceName" version="Version">
        <key1>value1</key1>
        <key2>value2</key2>
</OperationType>
```

Since only the XML contents (payloads) are of interest, HTTP POST requests are represented by their contents from now on.

WFS Response

Regardless of whether a WFS request is encoded KVP or XML, its response is always encoded in XML [Vre14]. Which contents these responses hold depends on their respective WFS requests. For instance, if the request is a transaction operation, the response shall be a confirmation from the server stating whether said operation is successful or failed. On the other hand, if the request is a query operation, the found features shall be returned as GML elements nested in an XML parent.

2.6.2 WFS Operations

In order to cover a variety number of requests issued by clients, the WFS standard defines a total of eleven operations, which can be divided into five major categories, namely the discovery, query, locking, transaction and stored query operations [Vre14], which are listed as follows:

- Discovery operations:
  - `GetCapabilities`
  - `DescribeFeatureType`
- Query operations:
  - `GetPropertyValue`
  - `GetFeature`
  - `GetFeatureWithLock` (also a locking operation)
- Locking operations:
  - `GetFeatureWithLock` (also a query operation)
  - `LockFeature`
- Transaction operation:
  - `Transaction`
- Stored query operations:
  - `CreateStoredQuery`
  - `DropStoredQuery`
  - `ListStoredQueries`
  - `DescribeStoredQueries`

Table 2.1 shows an overview of WFS operations including their operation groups (indicated by colours), descriptions and request encodings. For a more comprehensive look at the syntax of their requests as well as responses, please refer to [Vre14].

It is however not compulsory for a WFS to support all the above-mentioned operations. In fact, [Vre14] defines four different versions with increasing complexity and functionalities, namely “Simple WFS”, “Basic WFS”, “Transactional WFS” and “Locking WFS” (see Table 2.2).

2.6.3 WFS for CityGML

Since CityGML is a GML application schema, it can be transported and manipulated via a WFS that conforms with the previously introduced OGC international standard [Vre14]. In particular, the 3D City Database WFS version allows direct access to 3D city objects stored in a database by using platform and database independent calls. Therefore, not only do users no longer have to solely rely on the 3D City Database Importer/Exporter tool for data retrieval, they are also able to develop various CityGML-aware applications that make use of the interface [3DC16].

The 3D City Database WFS conforms with the second version of the OGC WFS international standard. In fact, it satisfies the “Simple WFS” conformance class defined in [Vre14]. The development of this WFS is currently led by the company virtualcitySYSTEMS GmbH, which offers its own vendor-specific extension called “virtualcityWFS”. This interface extends the basic 3D City Database WFS with additional functionalities, such as thematic and spatial filter capabilities and transaction support (e.g. insertion of CityGML complex (generic) feature properties) [3DC16; vir16]. Table 2.2 shows the differences in supported operations between various WFS versions.

	Operation	Description	Encoding	
			KVP	XML
🟦	`GetCapabilities`	Generates a service metadata document describing a WFS service provided by a server.	✔	✔
🟦	`DescribeFeatureType`	Returns a schema description of feature types offered by a WFS instance.	✔	✔
🟩	`GetPropertyValue`	Allows the value of a feature property or part of the value of a complex feature property to be retrieved from the data store for a set of features identified using a query expression.	✔	✔
🟩	`GetFeature`	Returns a selection of features from a data store using a query expression.	✔	✔
⬛	`GetFeatureWithLock`	Is functionally similar to the `GetFeature` operation except that in response to a `GetFeatureWithLock` operation, the WFS shall also lock the features in the result set.	✔	✔
🟥	`LockFeature`	Exposes a long-term feature locking mechanism to ensure data consistency in case of concurrent data transformation operations (e.g., update or delete).	✔	✔
🟨	`Transaction`	Describes data manipulation operations to be applied to feature instances under the control of a WFS.		✔
▨	`CreateStoredQuery`	Creates a stored query.		✔
▨	`DropStoredQuery`	Allows previously created stored queries to be dropped from the system.	✔	✔
▨	`ListStoredQueries`	Lists the stored queries available at a server.	✔	✔
▨	`DescribeStoredQueries`	Provides detailed metadata about each stored query expression that a server offers.	✔	✔

🟦	Discovery operations	🟩	Query operations	⬛	Query/Locking operation
🟥	Locking operations	🟨	Transaction operation	▨	Stored query operations

Table 2.1: An overview of WFS operations including their operation groups (indicated by colours), descriptions and request encodings. Adapted from [Vre14].

Operation	**Simple WFS**	**Basic WFS**	**WFS-T**	**Locking WFS**	**3DCityDB WFS**	**VCS WFS**
`GetCapabilities`	✔	✔	✔	✔	✔	✔
`DescribeFeatureType`	✔	✔	✔	✔	✔	✔
`GetPropertyValue`		✔	✔	✔		✔
`GetFeature`	✔[1]	✔[3]	✔[3]	✔[3]	✔[1]	✔[5]
`GetFeatureWithLock`				✔[4]		
`LockFeature`				✔[4]		
`Transaction`			✔	✔		✔[6,7]
`CreateStoredQuery`*						✔
`DropStoredQuery`*						✔
`ListStoredQueries`	✔[2]	✔[2]	✔[2]	✔[2]	✔[2]	✔
`DescribeStoredQueries`	✔	✔	✔	✔	✔	✔

Notes:
[1] Only the `StoredQuery` action is allowed.
[2] One stored query that fetches a feature using its ID shall be available.
[3] The `Query` action is allowed.
[4] Either the `GetFeatureWithLock` or `LockFeature` operations shall be available.
[5] Ad-hoc queries and filter encoding are also supported.
[6] `Insert`, `Delete` and `Update` (not `Replace`) are supported.
[7] Additional `InsertComplexProperty` (VCS-specific extension) are enabled.
* These operations shall be implemented in the "Manage stored queries" WFS.

Table 2.2: WFS versions and their implemented operations. The symbol ✔ indicates a supported operation in the respective WFS version. The vendor-specific extension "virtualcityWFS" (shown as "VCS WFS") offers more comprehensive operation capabilities compared to the ordinary WFS. Adapted from [Vre14; 3DC16; vir16].

3 Mapping 3D City Models in CityGML onto a Graph Database

The first major step in the comparison of two 3D city models encoded in CityGML is to map their instance documents onto a graph database, as graph databases like Neo4j do not support XML as its underlying data structure. In order to enable automatic mapping of arbitrarily large-sized CityGML datasets and to make preparations for matching in the next chapter, the mapping process is divided into the following smaller steps:

1. Reading CityGML datasets in Java,
2. Converting Java objects to graph entities,
3. Connecting mapped city objects using XLinks,
4. Calculating minimum bounding boxes of mapped city objects.

Steps 1 and 2 ensure arbitrarily large-sized input datasets can be completely read into a graph database efficiently without heavily relying on available main memory capacity. Java is used as the main programming language due to the fact that both citygml4j and Neo4j are built in Java (as explained in Sections 2.3 and 2.4 respectively). Step 3 connects mapped graph components together using XLinks (or `hrefs`). The result is a complete and connected graph representation of a respective CityGML instance document. Step 4 makes preparations for the matching process so that two graphs can be compared efficiently. Figures 3.1 and 3.2 give a graphical overview of the mapping process.

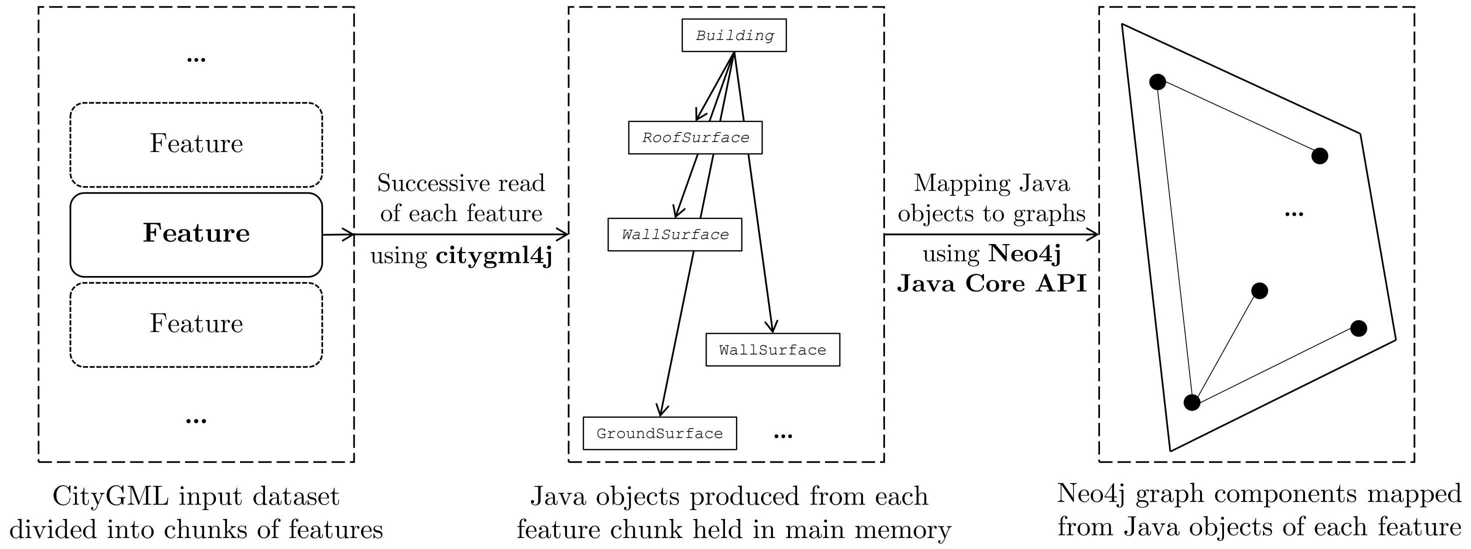

...
Feature
Feature
Feature
...
Successive read of each feature
using **citygml4j**
Building
RoofSurface
WallSurface
WallSurface
GroundSurface
...
Mapping Java objects to graphs
using **Neo4j Java Core API**
...
CityGML input dataset divided into chunks of features
Java objects produced from each feature chunk held in main memory
Neo4j graph components mapped from Java objects of each feature

Figure 3.1: An illustration of Step 1 (unmarshalling CityGML documents) and Step 2 (mapping Java objects onto graphs) of the mapping process.

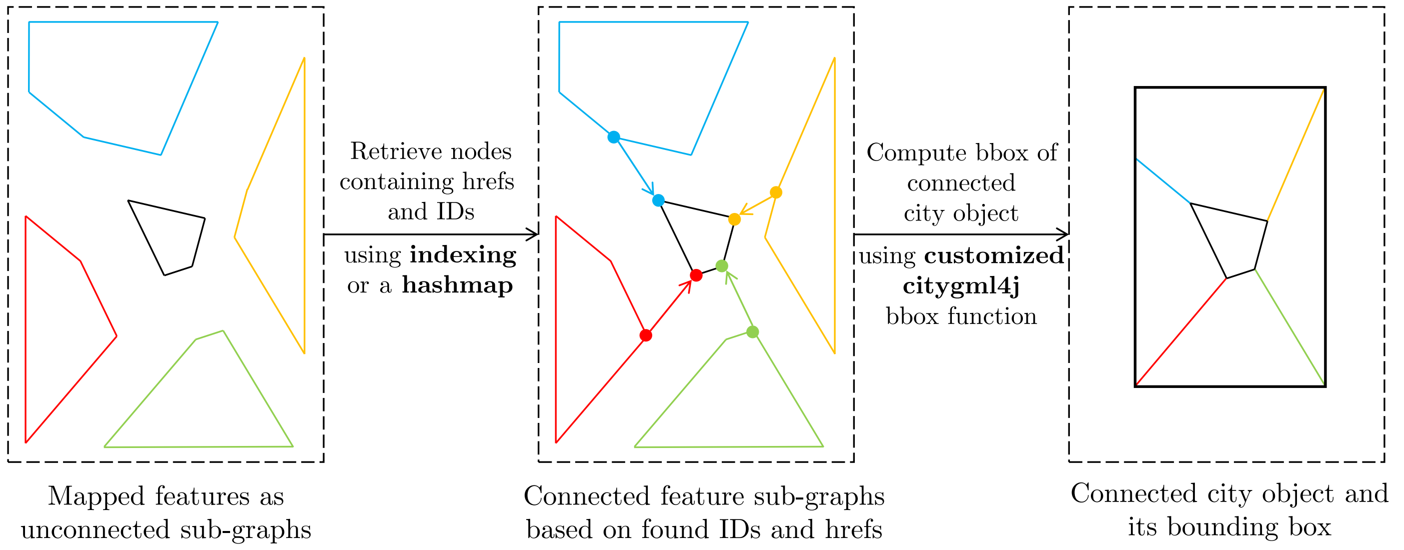

Retrieve nodes containing hrefs and IDs
using **indexing** or a **hashmap**
Compute bbox of connected city object
using **customized citygml4j** bbox function
Mapped features as unconnected sub-graphs
Connected feature sub-graphs based on found IDs and hrefs
Connected city object and its bounding box

Figure 3.2: An illustration of Step 3 (resolving XLinks) and Step 4 (computing minimum bounding boxes as a preparation for matching in the next chapter) of the mapping process.

3.1 Reading CityGML Datasets in Java

As explained in Sections 2.2 and 2.3, CityGML documents can be processed with the help of various XML parsing APIs in Java such as the DOM, SAX, StAX or JAXB. Each API comes with their own advantages and disadvantages over the others with respect to the application domains.

To determine which XML APIs are applicable in the context of reading and later comparing 3D city models encoded in CityGML, the following three factors are to be considered:

1. CityGML datasets can grow quickly in size (e.g. approximately 20 GB of geographic information as in the 3D city model of Berlin);

2. Flexible navigation between loaded objects must be enabled;

3. An object-oriented view of read CityGML data must be maintained, as not only stored data but how they are associated with each other play a central role in the implementation of the mapping process.

The first factor excludes the sole use of the DOM or JAXB because loading an entire large document completely into main memory is technically inefficient and might even be impossible for huge files. The second factor rules out the SAX, as it only allows navigation in one direction only. The third factor disqualifies the StAX, since, despite its capability to navigate well through XML documents, it does not provide an object-oriented view of read data. Thus, in order to fulfil all above-mentioned requirements, a combination of JAXB and SAX (or StAX) is proposed. By combining the advantages of these APIs, this approach allows partial unmarshalling (or deserialization) of XML data to Java objects and vice versa with efficient main memory consumption (as an advantage of SAX). In addition, it also offers an object-oriented view of mapped Java objects and their associations (as an advantage of JAXB).

In fact, the library **citygml4j** makes extensive use of this combined approach to read and handle CityGML datasets. As illustrated in Figures 2.6 and 3.1, the process of reading arbitrarily sized CityGML instance documents into Java objects is summarized as follows [SN09; Nag17]:

1. Based on a provided XML schema of the underlying CityGML data structure, the JAXB binding compiler creates Java model classes accordingly;

2. The SAX (or StAX) controller divides CityGML documents into a series of chunks (or pieces), each of which contains a (top-level) feature;

3. The JAXB sequentially unmarshals (or deserializes) each chunk completely into Java objects. As CityGML documents conform with given XML schema, mapped Java objects are instances of their respective model classes created in the previous step. Moreover, since the produced Java objects are held in main memory, full access to instance attributes and their associations is possible.

Operations can then be executed on mapped Java objects (which shall be explained in the next steps) before they are eventually flushed and replaced, as new chunks are unmarshalled into main memory.

3.2 Converting Java Objects to Graph Entities

This step takes place directly after each CityGML chunk has been unmarshalled into Java objects as described previously. Its objective is to transform these Java objects to corresponding graph entities in Neo4j conserving as much data as possible (see Figure 3.1). For this purpose, the **Neo4j Java Core API** introduced in Section 2.4.5 is employed. Conceptually however, two major challenges arise:

- Firstly, unmarshalled Java instances belong to a very complex class hierarchy defined by the XML schema of CityGML. This poses the difficulty in designing a suitable graph structure, such as how to efficiently represent instances of the same superclass in such a meaningful way that their hierarchical information can be made use of. Straightforward mapping of too much detail of the class hierarchy would lead to unnecessary node redundancies. On the contrary, too little information on the class hierarchy results in type ambiguities and data losses;

- Secondly, Neo4j is a value-based graph database, which means that no explicit schema modelling is possible. As a result, it is difficult to map Java objects to graph entities without losing any information, especially their hierarchical relations. Thus, it is utterly important to design an efficient but expressive enough graph structure that can capture and store as much of such information from Java objects as possible.

To resolve these challenges, three approaches have been developed throughout the course of this research, namely the instance-based approach, the hierarchy-based approach, and a combination of these two. Although eventually, they all effectively produce graph representations of input CityGML datasets, the difference lies in their handling of Java instances and the structure of respective created graphs.

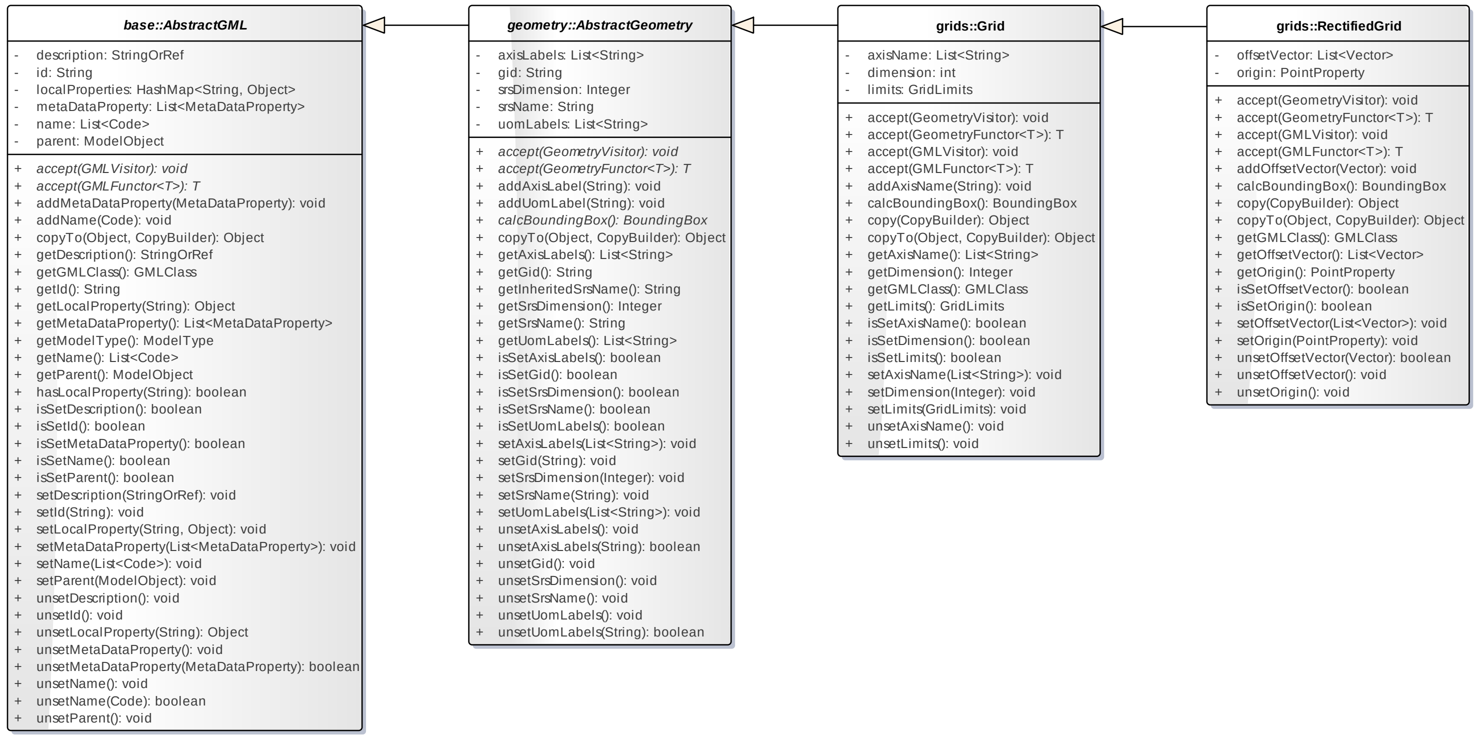

base::AbstractGML
- description: StringOrRef
- id: String
- localProperties: HashMap<String, Object>
- metaDataProperty: List<MetaDataProperty>
- name: List<Code>
- parent: ModelObject
+ accept(GMLVisitor): void
+ accept(GMLFunctor<T>): T
+ addMetaDataProperty(MetaDataProperty): void
+ addName(Code): void
+ copyTo(Object, CopyBuilder): Object
+ getDescription(): StringOrRef
+ getGMLClass(): GMLClass
+ getId(): String
+ getLocalProperty(String): Object
+ getMetaDataProperty(): List<MetaDataProperty>
+ getModelType(): ModelType
+ getName(): List<Code>
+ getParent(): ModelObject
+ hasLocalProperty(String): boolean
+ isSetDescription(): boolean
+ isSetId(): boolean
+ isSetMetaDataProperty(): boolean
+ isSetName(): boolean
+ isSetParent(): boolean
+ setDescription(StringOrRef): void
+ setId(String): void
+ setLocalProperty(String, Object): void
+ setMetaDataProperty(List<MetaDataProperty>): void
+ setName(List<Code>): void
+ setParent(ModelObject): void
+ unsetDescription(): void
+ unsetId(): void
+ unsetLocalProperty(String): Object
+ unsetMetaDataProperty(): void
+ unsetMetaDataProperty(MetaDataProperty): boolean
+ unsetName(): void
+ unsetName(Code): boolean
+ unsetParent(): void
geometry::AbstractGeometry
- axisLabels: List<String>
- gid: String
- srsDimension: Integer
- srsName: String
- uomLabels: List<String>
+ accept(GeometryVisitor): void
+ accept(GeometryFunctor<T>): T
+ addAxisLabel(String): void
+ addUomLabel(String): void
+ calcBoundingBox(): BoundingBox
+ copyTo(Object, CopyBuilder): Object
+ getAxisLabels(): List<String>
+ getGid(): String
+ getInheritedSrsName(): String
+ getSrsDimension(): Integer
+ getSrsName(): String
+ getUomLabels(): List<String>
+ isSetAxisLabels(): boolean
+ isSetGid(): boolean
+ isSetSrsDimension(): boolean
+ isSetSrsName(): boolean
+ isSetUomLabels(): boolean
+ setAxisLabels(List<String>): void
+ setGid(String): void
+ setSrsDimension(Integer): void
+ setSrsName(String): void
+ setUomLabels(List<String>): void
+ unsetAxisLabels(): void
+ unsetAxisLabels(String): boolean
+ unsetGid(): void
+ unsetSrsDimension(): void
+ unsetSrsName(): void
+ unsetUomLabels(): void
+ unsetUomLabels(String): boolean
grids::Grid
- axisName: List<String>
- dimension: int
- limits: GridLimits
+ accept(GeometryVisitor): void
+ accept(GeometryFunctor<T>): T
+ accept(GMLVisitor): void
+ accept(GMLFunctor<T>): T
+ addAxisName(String): void
+ calcBoundingBox(): BoundingBox
+ copy(CopyBuilder): Object
+ copyTo(Object, CopyBuilder): Object
+ getAxisName(): List<String>
+ getDimension(): Integer
+ getGMLClass(): GMLClass
+ getLimits(): GridLimits
+ isSetAxisName(): boolean
+ isSetDimension(): boolean
+ isSetLimits(): boolean
+ setAxisName(List<String>): void
+ setDimension(Integer): void
+ setLimits(GridLimits): void
+ unsetAxisName(): void
+ unsetLimits(): void
grids::RectifiedGrid
- offsetVector: List<Vector>
- origin: PointProperty
+ accept(GeometryVisitor): void
+ accept(GeometryFunctor<T>): T
+ accept(GMLVisitor): void
+ accept(GMLFunctor<T>): T
+ addOffsetVector(Vector): void
+ calcBoundingBox(): BoundingBox
+ copy(CopyBuilder): Object
+ copyTo(Object, CopyBuilder): Object
+ getGMLClass(): GMLClass
+ getOffsetVector(): List<Vector>
+ getOrigin(): PointProperty
+ isSetOffsetVector(): boolean
+ isSetOrigin(): boolean
+ setOffsetVector(List<Vector>): void
+ setOrigin(PointProperty): void
+ unsetOffsetVector(Vector): boolean
+ unsetOffsetVector(): void
+ unsetOrigin(): void

Figure 3.3: UML class diagram of `RectifiedGrid` and its superclasses `Grid`, `AbstractGeometry` and `AbstractGML` from Java model classes bound by JAXB and XML schema of CityGML in citygml4j.

3.2.1 Instance-based Approach

This approach focuses on the fact that Neo4j is a value-based graph database. In other words, it treats Java objects separately as sole data sources. The mapping concept of this approach is summarized in Algorithm 1.

Algorithm 1: `instance_based_map(instance)`

```
    Input  : A Java instance
    Output: Created node in graph database
 1  create a node in graph database;
 2  set node.label equal to instance.className;
 3  initialize attributes as a set of all available attributes and references of instance;
 4  foreach attribute of attributes do
 5      if attribute can be stored as simple texts then
 6          store attribute as a property in node;
 7      else
 8          create a child node by calling instance_based_map(attribute);
 9          create a relationship from node to child;
10      end
11  end
12  return node;
```

Explanatory notes:

Line 1 A node is created using the Neo4j Java Core API;

Line 2 Each created node has a label indicating the type or class name of their respective original Java objects;

Line 3 All available (i.e. own and inherited) attributes and references of the Java object are retrieved (e.g. with the help of the so-called `getter` methods);

Lines 5 and 6 A simple attribute that can be stored as plain text without losing any information (such as booleans, numbers, strings, etc.) is written to the current node as its property. The condition in Line 5 may vary between different implementations.

Lines 8 and 9 A complex reference (such as an object link to other instances) cannot be described as plain texts and thus must be represented as another node (Line 8) and a relationship between two nodes (Line 9). How this node is created depends on specific implementations, for example via recursive calls or overloaded functions. If a recursive method is used, it must be able to distinguish between different types of Java classes. On the other hand, if a number of overloaded functions are employed, then each of them must handle one type of classes. The process is repeated until all Java objects have been mapped.

Consider an example of class `RectifiedGrid` and its super classes `Grid`, `AbstractGeometry` and `AbstractGML` extracted from the Java model classes created by the JAXB using the XML schema of CityGML in citygml4j as illustrated in Figure 3.3. Classes `RectifiedGrid` and `Grid` are both instantiable, while `AbstractGeometry` and `AbstractGML` are abstract and thus not instantiable. Additionally in this example, two Java instances of `Grid` and `RectifiedGrid` are to be mapped to a graph database.

According to Algorithm 1, for the first `Grid` object, a node labelled `GRID` is created. Searching through its available attributes and references with the help of visible `getter` methods (Line 3 of Algorithm 1), a table containing these values sorted by originating class is built, which is shown in Table 3.1[1]. Depending on their data types, node properties or additional child nodes are created accordingly (Lines 5 - 10). The exemplary end result of this current node is illustrated in Figure 3.4.

Inherited from class `AbstractGML`	**Inherited from class** `AbstractGeometry`	**Declared in class** `Grid`
`description`	`axisLabels`	`axisName`
`id`	`gid`	`dimension`
`metaDataProperty`	`srsDimension`	`limits`
`name`	`srsName`	
	`uomLabels`	

Table 3.1: Visible attributes and references of a `Grid` instance.

This approach produces intuitive results in a straightforward manner. Moreover, the created graph captures all available information stored in Java objects while still remaining compact. The drawback lies however in the technical difficulties encountered during the implementation phase, namely: In order to achieve such compact but expressive end results, Line 3 in Algorithm 1 in particular must be implemented specifically for each possible class type, as this approach only concentrates on individual Java classes and not how they are connected to each other.

[1] In citygml4j, it is not allowed to retrieve the entire hash map set of `localProperties` in class `AbstractGML` without knowing its key values beforehand. For the sake of simplicity, the reference `parent` is omitted.

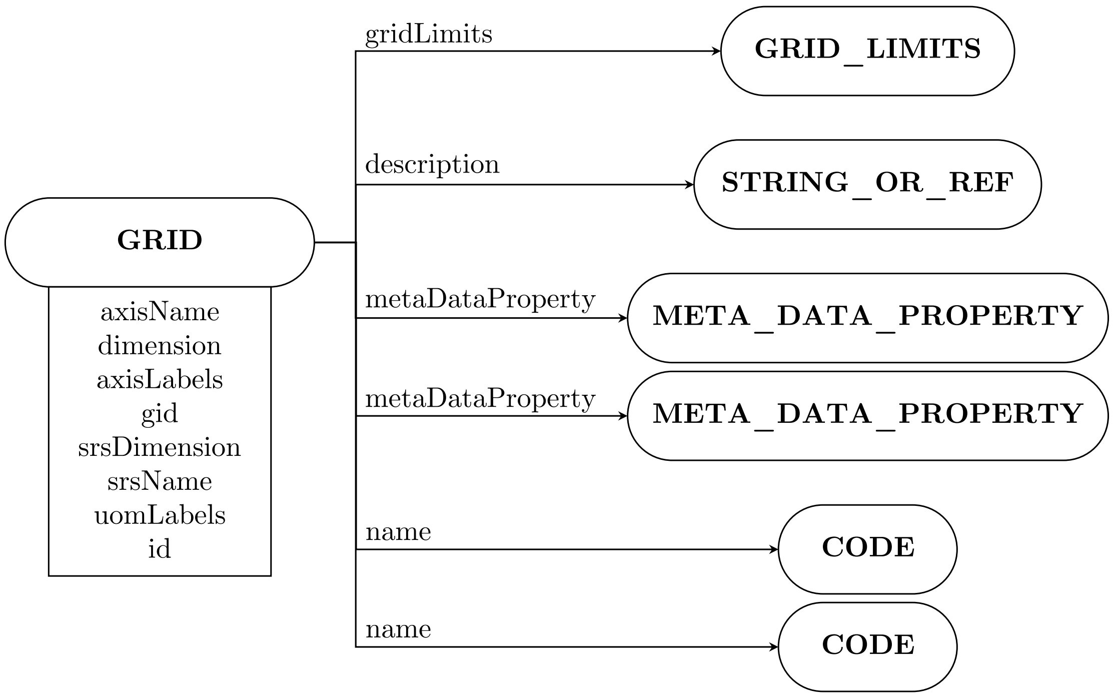


Figure 3.4: Result of mapping a `Grid` instance to a graph in the instance-based approach. Each rounded rectangle is a node, while (directed) arrows represent relationships between them.

To illustrate this, consider the remaining `RectifiedGrid` instance in the example above. To deliver values as shown in Table 3.2, the implementation of Line 3 in Algorithm 1 searches through the entirety of `RectifiedGrid` and lists all found object values via `getters` without knowing that the same workload performed while mapping a `Grid` instance shown previously has also been executed. To be more specific, despite the first three columns of Table 3.2 being the same as those of Table 3.1, they are achieved by separate implementations. This leads to code redundancies. The deeper the class hierarchy is, the more severe these redundancies may become, to an extent that the disadvantages could outweigh the advantages.

Inherited from class `AbstractGML`	**Inherited from class** `AbstractGeometry`	**Inherited from class** `Grid`	**Declared in class** `RectifiedGrid`
`description`	`axisLabels`	`axisName`	offsetVector
`id`	`gid`	`dimension`	origin
`metaDataProperty`	`srsDimension`	`limits`	
`name`	`srsName`		
	`uomLabels`		

Table 3.2: Visible attributes and references of a `RectifiedGrid` instance.

A further explanation as why Line 3 in Algorithm 1 must be specifically implemented for each possible class type as mentioned previously is that in Java, it is not possible to programmatically extract all attributes of arbitrary objects. Even with the help of the `Reflection` library, it is still impossible to access attributes that are declared `private` in the super classes without violating their internal structures and encapsulation purposes. In other words, the available (public) `getter` methods are the only appropriate solution.

A summary of this approach's advantages and disadvantages is shown below:

- Advantages:
  - Intuitive approach,
  - Compact but expressive results;
- Disadvantages:
  - Code redundancies due to complex class hierarchy.

3.2.2 Hierarchy-based Approach

As opposed to the instance-based approach, the hierarchy-based approach focuses heavily on the relations between Java classes with respect to their inheritance hierarchy. The concept is summarized in Algorithm 2.

Algorithm 2: `hierarchy_based_map(instance)`

```
Input  : A Java instance
Output: Created sub-graph representing instance in a graph database
create a node in graph database;
set node.label equal to instance.className;
initialize attributes as a set of local attributes and references of instance;
foreach attribute of attributes do
    if attribute can be stored as simple texts then
        store attribute as a property in node;
    else
        create a child node by calling hierarchy_based_map(attribute);
        create a relationship from node to child;
    end
end
if instance inherits SuperClass then
    create a super node by calling hierarchy_based_map((SuperClass) instance);
    create a relationship INHERITS from node to super;
end
return node;
```

Explanatory notes:

Lines 1 and 2 These lines are functionally identical to Lines 1 and 2 of Algorithm 1.

Line 3 All **local** attributes and references (that are defined in the current class) of the Java object are retrieved. The use of `getter` methods is optional, since local attributes and references are always visible within their originating class.

Lines 5 - 10 These lines are functionally identical to Lines 5 - 10 of Algorithm 1.

Lines 12 - 15 As long as the current object still inherits some super class in a given hierarchy (Line 12), a new node shall be created to represent the contents of this super class. To achieve this, the same function `hierarchy_based_approach` is called but now with the current instance cast to its super class as input argument (Line 13). Similarly to the instance-based approach, how this node is created depends on specific implementations, for example via recursive calls or overloaded functions. If a recursive method is used, it must be able to distinguish between different types of Java classes. On the other hand, if a number of overloaded functions are employed, then each of them must handle one type of classes. Finally, a relationship called `INHERITS` between the current node and its "super" node is created indicating an inheritance relationship between the pair (Line 14).

Thus, the main differences between the hierarchy and instance-based approach are:

- During its call, the hierarchy-based mapping function considers only local attributes and references of given Java objects, while the instance-based function searches for all local and inherited attributes;
- The hierarchy-based approach additionally creates "super" nodes representing the contents inherited from the super classes of given objects. These nodes are referenced by the relationships `INHERITS`.

Consider the same example of class `RectifiedGrid` and its super classes `Grid`, `AbstractGeometry` and `AbstractGML` from Section 3.2.1 as illustrated in Figure 3.3. According to Algorithm 2, for the object of `Grid` type, a node labeled `GRID` is created in the first call. Searching through its local attributes and references (Line 3 of Algorithm 2), the third column of Table 3.1 is found. Similarly to the instance-based approach, node properties or additional child nodes are created depending on their data types, (Lines 5 - 10). Since class `Grid` inherits `AbstractGeometry` (Line 12), the function `hierarchy_based_map` is called the second time to create further nodes representing the contents of this super class (Line 13). Here, the second column of Table 3.1 is built. As `AbstractGML` is the super class of `AbstractGeometry`, additional nodes shall be created accordingly in the third function call, where the remaining column of Table 3.1 is filled. Finally, returning nodes of each class are connected via the `INHERITS` relationships accordingly (Line 14). The exemplary end results are illustrated in Figure 3.5.

The advantage of this approach is that the inheritance hierarchy of Java classes is represented explicitly in the value-based graph, as the chain of created nodes exposes which attributes are contributed by which class within the hierarchy. Additionally, since most functions can be recycled due to the explicit use of hierarchical modelling, the implementation of this approach is much more compact. For instance, consider the remaining `RectifiedGrid` object: Compared to the `Grid` instance, the mapping process requires only one additional function implementation for class `RectifiedGrid`, as other functions have already been implemented. Furthermore, since the handling of each class is implemented only once, possible behaviour changes of the mapping

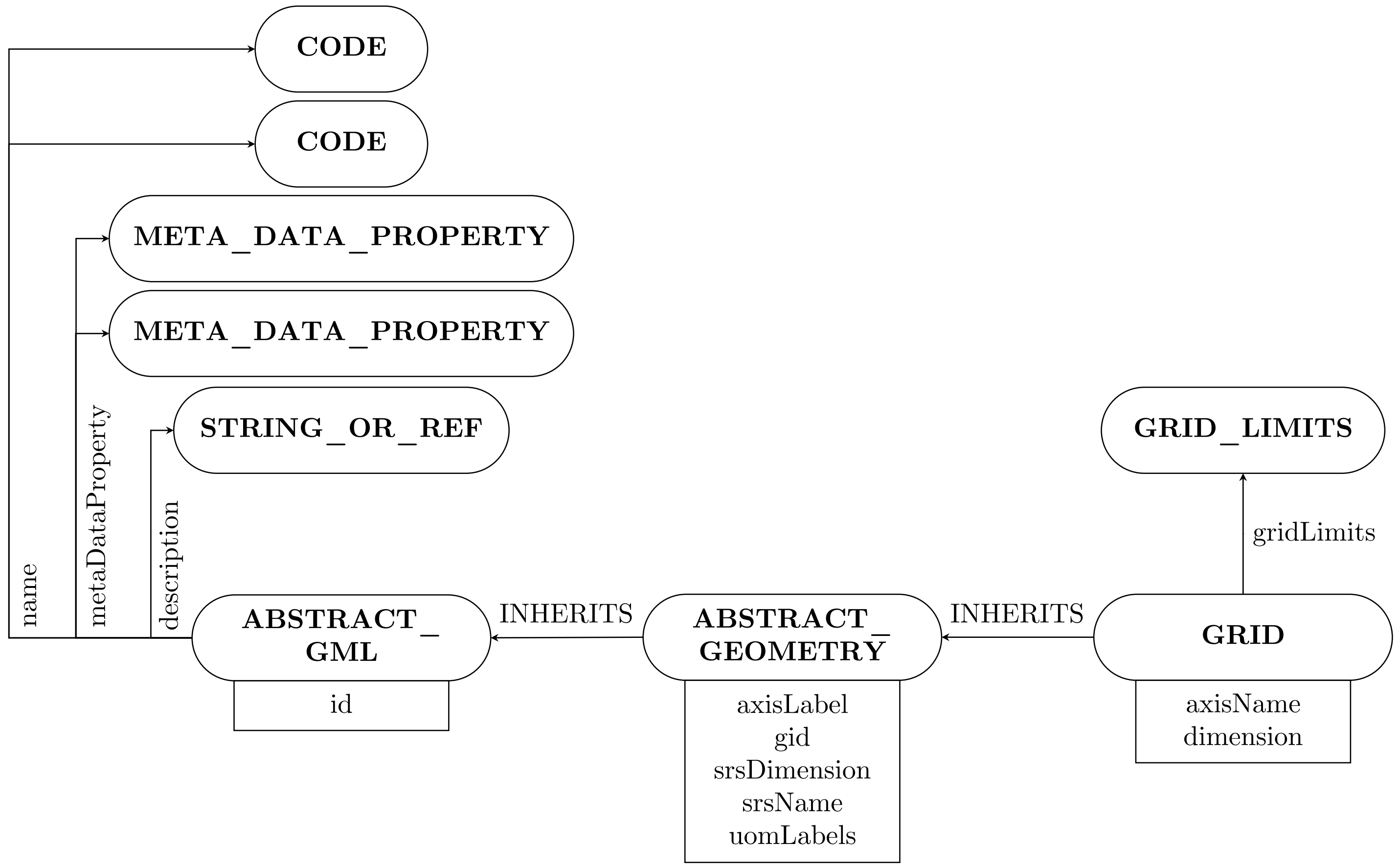


Figure 3.5: Result of mapping a `Grid` instance to a graph in the hierarchy-based approach. Each rounded rectangle is a node, while (directed) arrows represent relationships between them. The `INHERITS` relationships indicate inheritance between objects.

process against objects of these classes are easier to manage and maintain. However, this approach does produce a significantly larger number of nodes compared to the instance-based one. The deeper the class hierarchy is, the more nodes and `INHERITS` relationships shall be created. In addition, the fact that attributes and references of an object are stored in multiple nodes makes them more difficult to access from this object node's location. For example, in Figure 3.5, although the node labelled `STRING_OR_REF` is connected to an `ABSTRACT_GML` node via the `description` relationship, it semantically belongs to the `GRID` node on the right hand side. Thus, to travel between these two locations, three relationships (one `description` and two `INHERITS`) and two nodes (one `ABSTRACT_GML` and one `ABSTRACT_GEOMETRY`) are traversed, which generally complicates the whole query process in such mapped graphs.

A summary of this approach's advantages and disadvantages is shown below:

- Advantages:
  - A clear object-oriented view of value-based nodes,
  - Efficient implementation;
- Disadvantages:
  - Significantly larger number of produced nodes,
  - Complicated traversing between mapped nodes.

3.2.3 Combination of Instance and Hierarchy-based Approach

While the instance-based approach treats individual objects as sole data sources, the hierarchy-based approach makes extensive use of the relations between objects with respect to their class hierarchy. They both have advantages and disadvantages as described previously. The third and final approach can be thought of as a median between these two opposites: It takes advantage of objects' relations and class hierarchy like the hierarchy-based, but produces a compact graph similar to that of the instance-based approach. An overview of its general concept is shown in Algorithm 3.

Explanatory notes:

Input parameter `container` A container node representing the entirety of a Java object. All local and inherited contents of the object are stored in or connected to this `container`.

Lines 1 - 5 If the `container` is not yet initialized (Line 1), a new node is created (Line 2) and labelled (Line 3). The `container` is then set to be this new node (Line 4).

Line 6 This line is functionally identical to Line 3 of Algorithm 2.

Algorithm 3: `hybrid_map(instance, container)`

```
   Input  : A Java instance
   Output: Created node in a graph database
1  if container is null then
2      create a node in graph database;
3      set node.label equal to instance.className;
4      set container equal to this node;
5  end
6  initialize attributes as a set of local attributes and references of instance;
7  foreach attribute of attributes do
8      if attribute can be stored as simple texts then
9          store attribute as a property in container;
10     else
11         create a child node by calling hybrid_map(attribute, null);
12         create a relationship from container to child;
13     end
14 end
15 if instance inherits SuperClass then
16     call hybrid_map((SuperClass) instance, container);
17 end
18 return container;
```

Lines 8 - 13 These lines have similar functionalities to Lines 5 - 10 of Algorithm 1 as well as Lines 5 - 10 of Algorithm 2. However, instead of `node`, the variable `container` is used. Furthermore, the function call in Line 11 is adjusted to have a new empty `container` as additional parameter.

Lines 15 - 17 These lines are functionally similar to Lines 12 - 15 of Algorithm 2. However, the function `hybrid_map` has been adjusted to have an additional parameter, which receives the value of the non-empty `container`. This means that in case of an existing super class, the current `container` is extended with its contents. Also, how this `container` is created and extended depends on specific implementations, for example via recursive calls, or overloaded functions. If a recursive method is used, it must be able to distinguish between different types of Java classes. On the other hand, if a number of overloaded functions are employed, then each of them must handle one type of classes.

The main difference between this "hybrid" and other approaches is the use of a central expandable `container` node, where all (i.e. own and inherited) attributes and references of respective Java object can be stored. Considering the previous example of four classes `RectifiedGrid`, `Grid`, `AbstractGeometry` and `AbstractGML` as shown in Figure 3.3, the following sequence of function calls illustrates the process of mapping a `Grid` instance in this hybrid approach:

1. Call `hybrid_map(instance, null)`:
    - A `GRID` node as a `container` is created,
    - `axisName` and `dimension` are added to the `container`'s properties,
    - A `GRID_LIMITS` child node is attached to `container`;
2. Call `hybrid_map((AbstractGeometry) instance, container)`:
    - `axisLabels`, `gid`, `srsDimension`, `srsName` and `uomLabels` are added to the `container`'s properties;
3. Call `hybrid_map((AbstractGML) instance, container)`:
    - `id` is added to the `container`'s properties,
    - Child nodes `STRING_OR_REF`, `META_DATA_PROPERTY` and `CODE` are attached to the `container`.

Therefore, the advantage of this hybrid approach is that it can achieve the same compact but expressive mapped graphs as those produced by the instance-based approach, while possessing the robust implementation based on hierarchical information as in the hierarchy-based approach. The drawback is the loss of explicit class hierarchy in created graphs. This is however acceptable, since hierarchical information is mainly needed for the implementation of the mapping process and generally does not play a central role in the comparison of value-based graphs.

A summary of this approach's advantages and disadvantages is shown below:

- Advantages:
    - Compact but expressive results;
    - Robust implementation;
- Disadvantages:
    - No explicit information about class hierarchies within created graphs.

Hence, to map Java objects to graph entities, the hybrid approach is employed in the implementation of this research.

3.3 Connecting Mapped City Objects using XLinks

3.3.1 Existence of XLinks in Mapped Graphs

Severed Features Due to Partial Unmarshalling

Section 3.1 addresses the problems of parsing large CityGML documents regarding the main memory consumption and proposes an approach dividing them into smaller feature chunks with the help of JAXB and SAX (or StAX) in citygml4j. Each chunk is then separately unmarshalled and consequently mapped onto sub-graphs in Neo4j as described in Section 3.2. As a result, the explicit connection between split features and their respective parent elements are lost during the process. Broadly speaking, each created sub-graph representation of split features can be thought of as a graph partition or an isolated region, of which the entire graph database consists. Thus, the first objective of this section is to reconstruct lost connections between such isolated regions and their respective parents as illustrated in Figure 3.2.

To enable the recovery of severed connections, before splitting, the library citygml4j looks for the ID of affected feature. If none is available, a new unique one shall automatically be generated. This ID is then stored in an XLink or `href` node, to which the relationship between aforementioned feature element and its respective parent now references (see Figure 3.6). This implicit information suffices for the connection reconstruction but creates a large number of new XLink references.

Syntactic Ambiguities Between In-line and XLink Declarations

XLink is a simple yet practical means to reusing existing elements without having to define them "in-line" repeatedly and thus reduces redundancies in XML documents [Bra+08; DeR+10]. However, despite their syntactic differences, both XLink and in-line declaration can be used to effectively define the same object. In order to match two CityGML instance documents correctly and efficiently, such syntactic ambiguities must be taken into account. Therefore, the second and last objective of this section is to transform objects defined in-line or by XLink to an unambiguous graph representation.

3.3.2 Resolving XLinks within the Graph Database

Both above-mentioned objectives can be achieved by resolving XLink or `href` nodes in a graph database, which can be realized in two different approaches using:

- A self-developed indexing mechanism with internal hash maps held in main memory;
- Built-in indices available in Neo4j stored on disk.

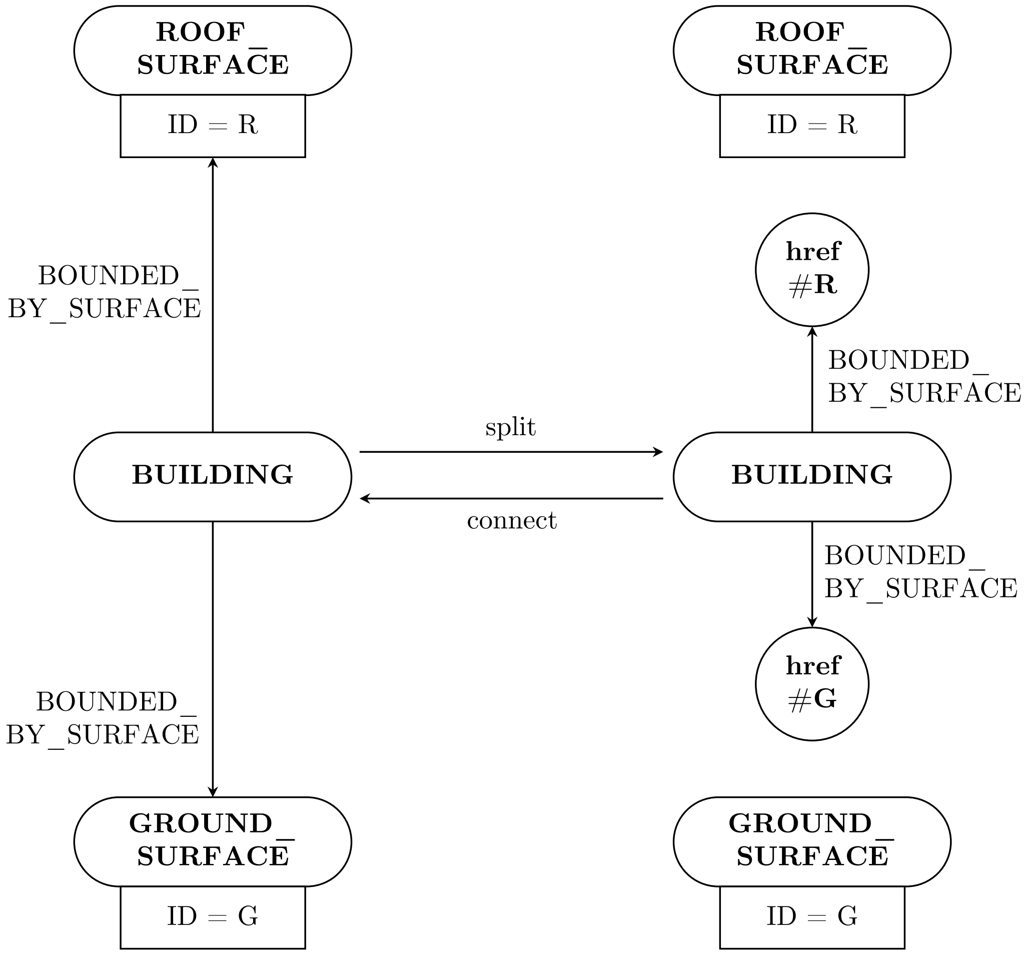


Figure 3.6: An illustration of the splitting mechanism per feature in citygml4j. Associations to features from parent elements are replaced by corresponding XLinks or `hrefs`, which can be used to connect split features back to their original parents.

Resolving XLink Using Internal Hash Maps

Nodes containing IDs and `hrefs` are stored in two different hash maps per city model. In Java, the `ConcurrentHashMap` instances can be employed as they support concurrency of data retrievals and updates. Each time a node containing an ID is encountered during the mapping process, a hash map entry consisting of this ID as key and a reference to the node as value is generated on the fly. The `href` hash map is filled in the similar manner. However, their major difference lies in the uniqueness of their keys:

- IDs are unique within a CityGML document, which means that each key of the ID hash map must occur at most once;
- An `href` can appear multiple times within a CityGML document. Hence, the structure of the `href` hash map must be adjusted accordingly so that multiple identical keys are allowed.

To resolve XLinks, each key stored in the `href` hash map is searched through the ID hash map. If a match is positive, both respective nodes are connected and the processed `href` is removed from the hash map. The process repeats until no more `href` remains.

The main advantage of using internal hash maps is their fast response time. This comes however at the cost of main memory consumption and thus may not be applicable in memory-limited systems.

Resolving XLinks Using Neo4j Indices

Alternatively, XLink nodes can be resolved by employing (manual or legacy) indices in Neo4j as described in Section 2.4.4. Similarly, two index sets for IDs and `hrefs` per city model are required, which basically also store nodes in a hash map structure. Based on specific implementations, multiple occurrences of `href` keys is possible in Neo4j.

The advantage of using Neo4j's built-in indices is the lower main memory consumption compared to the internal hash maps. However, this may slow down the mapping process due to costly disk read and write operations. Moreover, sufficient additional storage space must be reserved beforehand.

3.4 Calculating Minimum Bounding Boxes of Mapped City Objects

Unlike previous steps described in Sections 3.1 to 3.3, which are required to create a graph representation of a CityGML instance document, this last step (see Figure 3.2) serves as preparation for different strategies utilizing minimum bounding boxes of mapped city objects applied in the matching process in the next chapter.

3.4.1 Reverse-mapping Graphs to Java Objects

Ideally, in citygml4j, the minimum bounding box (i.e. minimum bounding rectangle in 2D or minimum bounding cuboid in 3D) of a spatial Java city object (e.g. `Building`) can be computed by a built-in function that takes all of its geometric contents (e.g. boundary surfaces) into account. However, this method has some limitations, namely:

- The function requires the input Java object to be completely available in main memory as a whole, which is not always the case, since Section 3.1 shows that large city objects are to be split into features;
- If the input Java object has an unresolvable XLink in runtime (e.g. XLink that belongs to another feature not yet loaded in main memory), the function may not be able to trace this reference.

By reversely transforming a sub-graph into its original Java object, the first limitation can be overcome. Moreover, since Section 3.3 has already resolved all XLinks and hence connected all sub-graphs, the second limitation will automatically be resolved, as reversely created Java objects are connected and do not contain XLinks.

Conceptually, the reverse mapping of graphs to Java objects is the opposite of the mapping process described in Section 3.2. However, they both are equally complex and play an important role in different aspects of the whole image: while the mapping process enables the creation of nodes and relationships from a given Java object and ultimately the matching of two CityGML datasets, the reverse mapping produces unambiguous Java objects from connected sub-graphs, which can then be marshalled back to CityGML. Due to the time constraint of this research, only the reverse mapping of Java objects of class `Building` and `BuildingPart` with their geometric properties (such as `RoofSurface`, `WallSurface`, etc.) is implemented.

3.4.2 Calculating Minimum Bounding Boxes of Java Objects

With the help of the reverse mapping process, a Java object (e.g. `Building`) created from a sub-graph can be passed to the previously mentioned built-in function in citygml4j to calculate its minimum bounding box. However, this step is only necessary if the sub-graph does not have the value of this minimum bounding box already (e.g. the association `boundedBy` of `BUILDING` nodes). Moreover, if no envelope is available for a city model, this value shall be successively computed on the fly from all minimum bounding boxes of its buildings.

4 Matching 3D City Models in CityGML using a Graph Database

This chapter describes the matching process of graphs mapped from CityGML datasets as described previously in Chapter 3. Since nodes play a central role in graphs, the matching process is based around the concept of their structure: Two graphs can be matched by recursively comparing the properties and relationships of all of their respective nodes. Thus, this chapter mainly focuses on matching nodes of provided graphs. The general concept of comparing two graphs starting with their root nodes is summarized in Algorithm 4.

Algorithm 4: `match_node(node1, node2)`

```
Input : node1 and node2 of graphs representing old and new city model resp.
1 match_properties(node1, node2);
2 match_relationships(node1, node2);
```

4.1 Comparing Node Properties

Actual data are mostly stored in node properties. Therefore, differences found in node properties indicate possible deviations of respective data sources. The concept of this step is shown in Algorithm 5.

Explanatory notes:

Lines 1 and 2 Property values stored in each node are retrieved. Depending on specific implementations, a hash map can be employed to hold both property names and their respective values.

Lines 3 - 10 Only values from the same property name (Line 3) are compared with one another. If both are equal (Line 4), depending on specific implementations, an action such as informing the matching process of this match may be taken (Line 5). Otherwise, an `UPDATE` operation is created indicating that the old value

Algorithm 5: `match_properties(node1, node2)`

```
   Input  : node1 and node2 of graphs representing old and new city model resp.
 1 let values1 be the set of all property values stored in node1;
 2 let values2 be the set of all property values stored in node2;
 3 foreach matched property_name of node1 and node2 do
 4     if values1.property_name = values2.property_name then
 5         inform matched property;
 6     else
 7         create an UPDATE operation;
 8     end
 9     remove property_name from node1 and node2;
10 end
11 foreach remaining property_name of node1 do
12     create a DELETE operation;
13 end
14 foreach remaining property_name of node2 do
15     create an INSERT operation;
16 end
```

of `property_name` has been changed in the new dataset (Line 7). An overview of all edit operations is shown in Section 5.1. Then, this matched property name is removed from both nodes. It is however recommended to use a temporary list storing property names of both nodes, so that removing a matched one does not interfere with the real contents of affected nodes.

Lines 11 - 13 Since `node1` belongs to the graph representing the older city model, the existence of unmatched properties (Line 11) indicates that they no longer exist in the newer city model. Thus, a `DELETE` operation is created for each of such properties.

Lines 14 - 16 An `INSERT` operation is created for each remaining properties in the graph representing the newer city model (Line 15).

Due to the fact that properties in Neo4j must be unique in each node, Algorithm 5 guarantees that all properties of both nodes are processed.

4.2 Matching Node Relationships

As opposed to the comparison of node properties described in Section 4.1, matching relationships between two given nodes is much more complex considering:

- In Neo4j, relationships can be traversed in both directions, namely `OUTGOING` and `INCOMING`. The matching process must however remain consistent in one specific direction while traversing through nodes, so that no node is processed twice. The chosen direction is `OUTGOING`, since the matching process starts with root nodes;
- A relationship has a relationship type and optional properties as explained in Section 2.4.4. The matching process uses only the former to distinguish and compare node relationships;
- In contrast to node properties in Neo4j, a relationship may occur multiple times for a given node. Its type however must be unique. For example, although a `BUILDING` node can have several relationships referencing its `BOUNDARY_SURFACE` child nodes, these relationships belong to a unique relationship type called `BOUNDED_BY_SURFACE`.

Taking these into account, Algorithm 6 describes the main concept of matching relationships of two given nodes, where:

Lines 1 - 19 Relationships of a matched type from both nodes are compared.

Lines 2 and 3 Since a node can theoretically have an arbitrary number of relationships from a single type and each relationship connects it with another node, matching relationships inevitably leads to comparing child nodes of the same relationship type. The matching process only traverses in `OUTGOING` direction as explained above, hence child nodes are considered.

Lines 4 - 11 For each child node from the first input node (representing an element from the older city model) (Line 4), the function `find_candidate` searches for a corresponding candidate node from the second list of the newer city model (Line 5) that it most likely matches. How this candidate is found depends of specific implementations and predefined rules. If a candidate is successfully found (Line 6), both two nodes are removed from their respective lists (Lines 7 and 8) and matched (Line 9). To avoid endless loops caused by cross-referencing, the mapping process ensures no directed circle exists in the graph database. Since the function `match_node` in Algorithm 4 also calls `match_relationships`, the whole process is considered recursive.

Lines 12 - 14 Child nodes from older city model that could not find a candidate shall be deleted (Line 23).

Algorithm 6: `match_relationships(node1, node2)`

```
Input   : node1 and node2 of graphs representing old and new city model resp.
foreach matched relationship_type of node1 and node2 do
    children1 ← node1.get_children_by_rel_type(relationship_type);
    children2 ← node2.get_children_by_rel_type(relationship_type);
    foreach child1 of children1 do
        child2 ← find_candidate(child1, children2);
        if child2 is not empty then
            remove child1 from children1;
            remove child2 from children2;
            match_node(child1, child2);
        end
    end
    foreach remaining child1 of children1 do
        create a DELETE operation;
    end
    foreach remaining child2 of children2 do
        create an INSERT operation;
    end
    remove relationship_type from node1 and node2;
end
foreach remaining relationship_type of node1 do
    children1 ← node1.get_children_by_rel_type(relationship_type);
    foreach child1 of children1 do
        create a DELETE operation;
    end
end
foreach remaining relationship_type of node2 do
    children2 ← node2.get_children_by_rel_type(relationship_type);
    foreach child2 of children2 do
        create an INSERT operation;
    end
end
```

- **Lines 15 - 17** Similarly, child nodes from newer city model that could not find a candidate shall be inserted (Line 16).
- **Line 18** The current relationship type is removed from both input nodes before a new one is loaded. It is recommended to store relationship types in a temporary list so that removing a relationship type does not interfere with actual data stored in the graph database.

Lines 20 - 25 Unprocessed relationship types remaining in older city model indicate that they no longer exist in the newer one. Hence, `DELETE` operations shall be created (Line 23).

Lines 26 - 31 Similarly, `INSERT` operations (Line 29) shall be created for all unprocessed child nodes remaining in the newer city model.

The function `find_candidate` in Line 5 of Algorithm 6 plays a decisive role in terms of both efficiency and correctness of the whole matching process, as it dictates which object pairs should be compared to one another. Depending on node types, a corresponding implementation that can distinguish individual objects based on their characteristics is required. In CityGML, the most important aspect that can be used as a matching pattern among objects is their geometrical properties. Hence, the following sections describe how geometries of most common objects can be taken into account while searching for the best matching candidate in their graph representations.

4.2.1 Matching Geometry of Points

Points are a primitive notion, upon which all other geometric objects are built. In citygml4j, a point can be represented by a number of various classes, which are listed with their respective node labels in Neo4j in Table 4.1. Regardless of such diversity in use cases, the geometric characteristics of a point remain unchanged: A point is defined by an n-tuple, where n is a non-negative integer and equal to the dimension of the underlying space. For instance, in one-dimensional domains (such as lines and segments), $n = 1$ and thus, a point has one coordinate. In two-dimensional space (such as areas), a point is represented by a 2-tuple (x, y), where x and y are its coordinates in each dimension. A three-dimensional point (x, y, z) can be defined in the similar manner. The most common point representations used in CityGML are two- and three-dimensional.

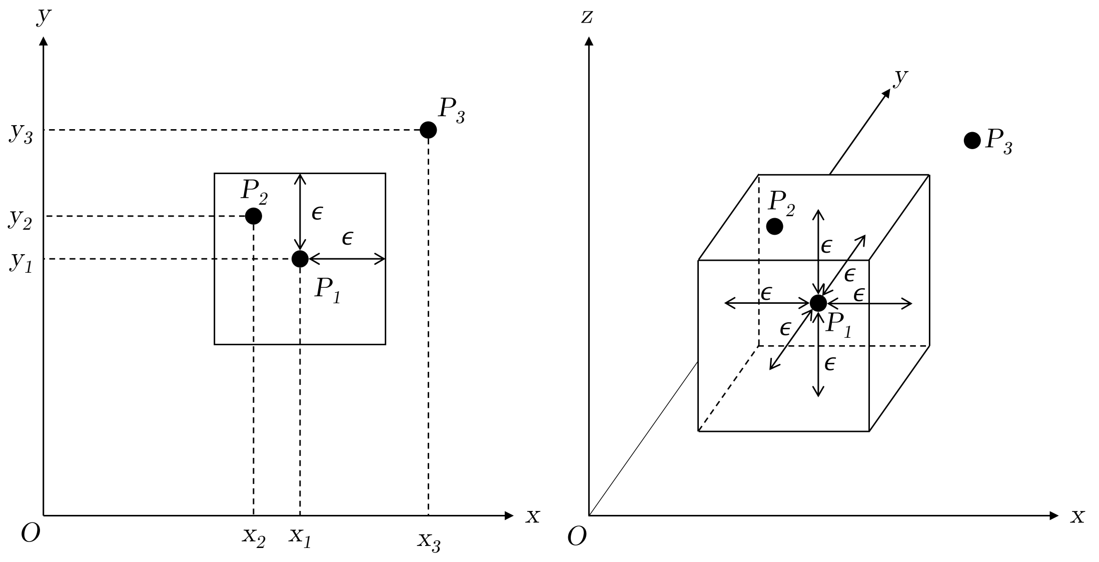
Since points do not have length, area or volume, the only property employed to distinguish them from others is their coordinates. In practice, however, coordinates of the same point location in real world may differ due to:

Class name in citygml4j	Node label in Neo4j
`Coord`	`COORD`
`Coordinates`	`COORDINATES`
`DirectPosition`	`DIRECT_POSITION`
`Point`	`POINT`
`PointProperty`	`POINT_PROPERTY`
`PosOrPointProperty-OrPointRep`	`POS_OR_POINT_PROPERTY-OR_POINT_REP`
`PosOrPointProperty-OrPointRepOrCoord`	`POS_OR_POINT_PROPERTY-OR_POINT_REP_OR_COORD`
`PointRep`	`POINT_REP`

Table 4.1: A list of point classes in citygml4j and their respective node labels in Neo4j.

- **Spatial Reference System (SRS) or Coordinate Reference System (CRS):** Since the earth's shape is an imperfect ellipsoid, a global reference system for all geographical entities leads to deviations in measured coordinates between locations. Thus, they are often recorded in different reference systems based on geographical locations called Spatial Reference System (SRS) or Coordinate Reference System (CRS) [Bil16]. Each SRS defines its own specific map projection as well as coordinate transformations between reference systems, which is out of scope of this research. Therefore, input CityGML instance documents must be provided in the same spatial reference system or a coordinate transformation must be performed beforehand to bring both documents to the same reference system;

- **Numeric and instrument errors:** On the other hand, even provided in one reference system, coordinates of two representations of the same point may still differ due to numerical (such as rounding) and instrument errors. Such minor deviations should be tolerated. Thus, for a reference point P_1 as centre, depending on the chosen distance indicator, a neighbourhood $N(\epsilon)$ is constructed, where ϵ is the maximum empirically predetermined allowed distance tolerance. For example, if the Euclidean distance indicator is chosen, $N(\epsilon)$ shall be a circle (2D) or a sphere (3D). However, to calculate this distance, expensive operations such as square roots and multiplications are required. Since the research focuses on matching 3D objects of massive datasets, for a small ϵ, it is often sufficient to compare coordinates in each dimension, which requires only subtractions. In this case, $N(\epsilon)$ shall be a square (2D) or a cube (3D) (see Figure 4.1). A point P_2 is geometrically matched with point P_1 if, and only if, P_2 is located inside of $N(\epsilon)$ of P_1 as formally described in Equation (4.1).



(a) $N(\epsilon)$ as a square in 2D. (b) $N(\epsilon)$ as a cube in 3D.

Figure 4.1: An illustration of the neighbourhood $N(\epsilon)$ of point P_1 in 2D (4.1a) and 3D (4.1b). Point P_2 is located inside $N(\epsilon)$ and therefore geometrically matched with P_1. Point P_3 on the other hand is located outside of $N(\epsilon)$ and thus cannot be matched with P_1.

$$\boxed{\begin{aligned} & P_2(x_2, y_2, z_2) \text{ is matched with } P_1(x_1, y_1, z_1) \\ \iff\; & P_2 \text{ is located inside } N(\epsilon) \\ \iff\; & \begin{cases} |x_2 - x_1| \le \epsilon \\ |y_2 - y_1| \le \epsilon \\ |z_2 - z_1| \le \epsilon. \end{cases} \end{aligned}} \tag{4.1}$$

Points geometrically matched by Equation (4.1) can be considered equal and thus no further comparison is required.

4.2.2 Matching Geometry of Line Segments

In CityGML, a (closed) line segment is a part of a one-dimensional straight line and is bounded by a start and end point called "control points", which can be represented as classes and labels in citygml4j and Neo4j respectively (see Table 4.1). A `LineString` is a set of consecutive line segments with linear interpolation in between. Thus, a

`LineString` can be defined by at least one pair of control points representing one segment. An end point of one segment must be a start point of the next one with the exceptions of the start point of the first segment and the end point of the last segment, which are allowed to be identical indicating the `LineString` is closed. Line segments within a `LineString` are unique and must not intersect each other.

Since line segments and `LineStrings` are composed of points, they can geometrically be matched using these points' coordinates. All consecutive collinear line segments (given an empirically predetermined distance tolerance) can be merged together and thus treated as a single segment during matching. As mentioned previously in Section 4.2.1, all coordinates must be transformed into the same reference system beforehand. Then, the geometries can be compared by iterating over all control points and examining their spatial similarities with error tolerance ϵ taken into account (see Equation (4.2) and Figure 4.2). `LineStrings` matched by Equation (4.2) can also be considered equal, which means that no further comparison is required.

$$\boxed{\begin{array}{l}\texttt{LineString}\ L_1 = (P_{11}, P_{12}, \ldots, P_{1n}) \text{ is matched with } L_2 = (P_{21}, P_{22}, \ldots, P_{2n}) \\ \iff \text{Point } P_{1i} \text{ of } L_1 \text{ is matched with } P_{2i} \text{ of } L_2\ \forall i \in [1, n]\,.\end{array}} \tag{4.2}$$

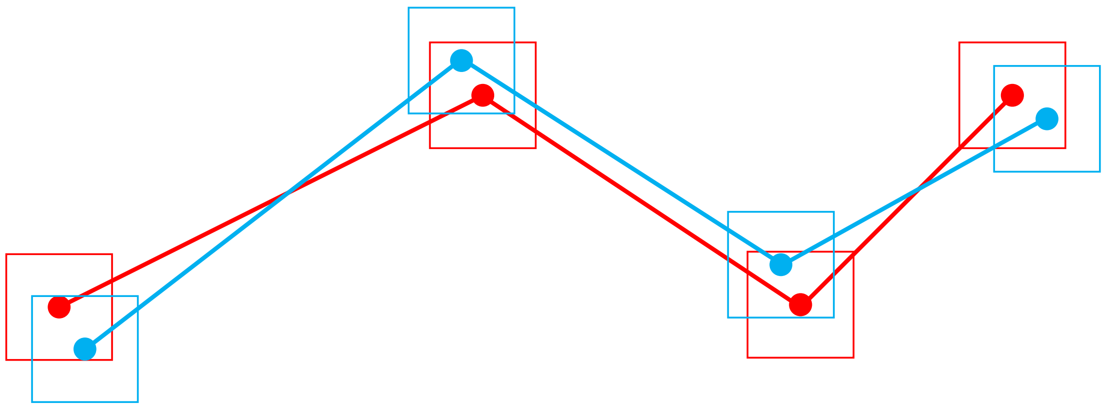


Figure 4.2: An example of two geometrically matched `LineStrings` (red and blue) each consisting of three line segments.

A more general concept of `LineStrings` are curves, each of whose curve segments may have a different interpolation method. A curve has a positive orientation. As long as such curves are composed of points, Equation (4.2) can be applied in a similar manner.

4.2.3 Matching Geometry of Rings

A ring in CityGML can be thought of as a closed `LineString` described in Section 4.2.2 (not as annulus as commonly called in mathematics). Buildings in CityGML make extensive use of polygons (Section 4.2.4), whose boundaries are often represented as `LinearRings`.

According to [Cox+04; Grö+12; Grö10], a finite sequence of points $R = (P_1, P_2, \ldots, P_n)$, $n \geq 4$, $P_i = (x_i, y_i, y_i)$ is a `LinearRing` if it fulfils the following requirements:

(i) The first and last point P_1 and P_n are identical (closeness).

(ii) All points except the first and last one are different:

$$P_i \not\equiv P_j \; \forall i, j \in [1, n-1], i \neq j.$$

(iii) Two arbitrary edges (P_i, P_{i+1}) and (P_j, P_{j+1}) with $i, j \in [1, n-1], i \neq j$ do not intersect one another at a start or end point. No self intersection exists.

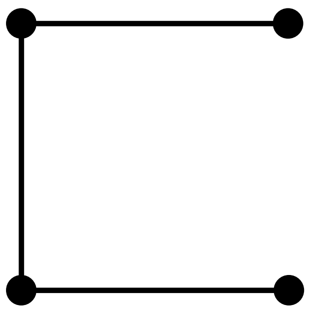
A `LinearRing` can theoretically consist of non-coplanar points. In the context of this research however, only planar `LinearRings` are considered. Figures 4.3a to 4.3d of Figure 4.3 show some examples that do not fulfil at least one of the above-mentioned requirements and thus are not qualified as `LinearRings`. On the other hand, the last example shown in Figure 4.3e satisfies all requirements and is therefore a `LinearRing`.

In order to match `LinearRings` (henceforth simply called rings) correctly, the following two geometrical properties are considered:

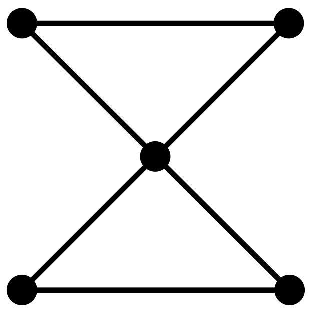
- Planar rings are arbitrarily oriented in three-dimensional space;
- Two geometrically equivalent rings can have different orders or numbers of points.

Rotating Rings to Reference Plane

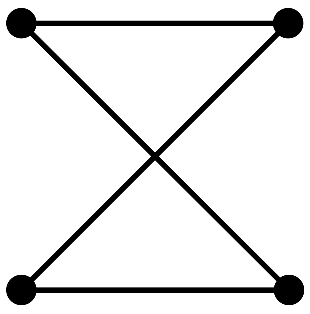
Despite being planar (as assumed previously), rings can still have arbitrary orientations in three-dimensional space. An example of two perpendicular rings is shown in Figure 4.4. Furthermore, most available geometry libraries such as the Java Abstract Window Toolkit (AWT) or Java Topology Suite (JTS) only function reliably in two-dimensional domains. Thus, normal vectors (or orientations) of 3D rings are first computed and compared. Only rings that have similar orientations and near-zero distance between their spanned planes (given an empirically predetermined angle and distance tolerance respectively) are to be rotated to a common reference plane (see description below). Otherwise, the process terminates as these rings have different orientations and are therefore geometrically unequal.



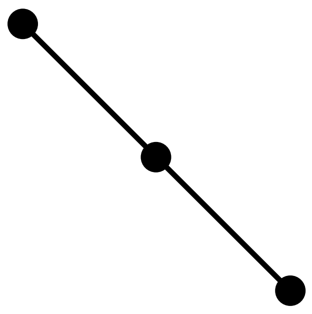
(a) Not a `LinearRing` due to a missing edge and thus not being closed.



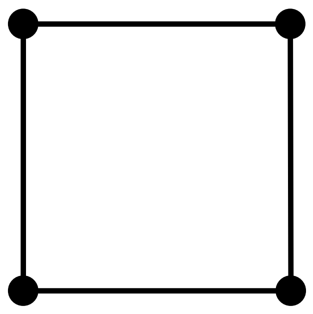
(b) Not a `LinearRing` due to the middle point occurring twice.



(c) Not a `LinearRing` due to self intersection of two diagonal edges.



(d) Not a `LinearRing` due to all collinear points.



(e) A correct `LinearRing` of four coplanar points.

Figure 4.3: Some examples illustrating the three requirements needed for `LinearRings`: closeness (Figure 4.3a), single occurrence of points (Figure 4.3b) and no self intersection (Figure 4.3c). Collinear points as shown in Figure 4.3d do not form a `LinearRing`. Figure 4.3e on the other hand shows a proper `LinearRing`. Adapted from [Grö10].

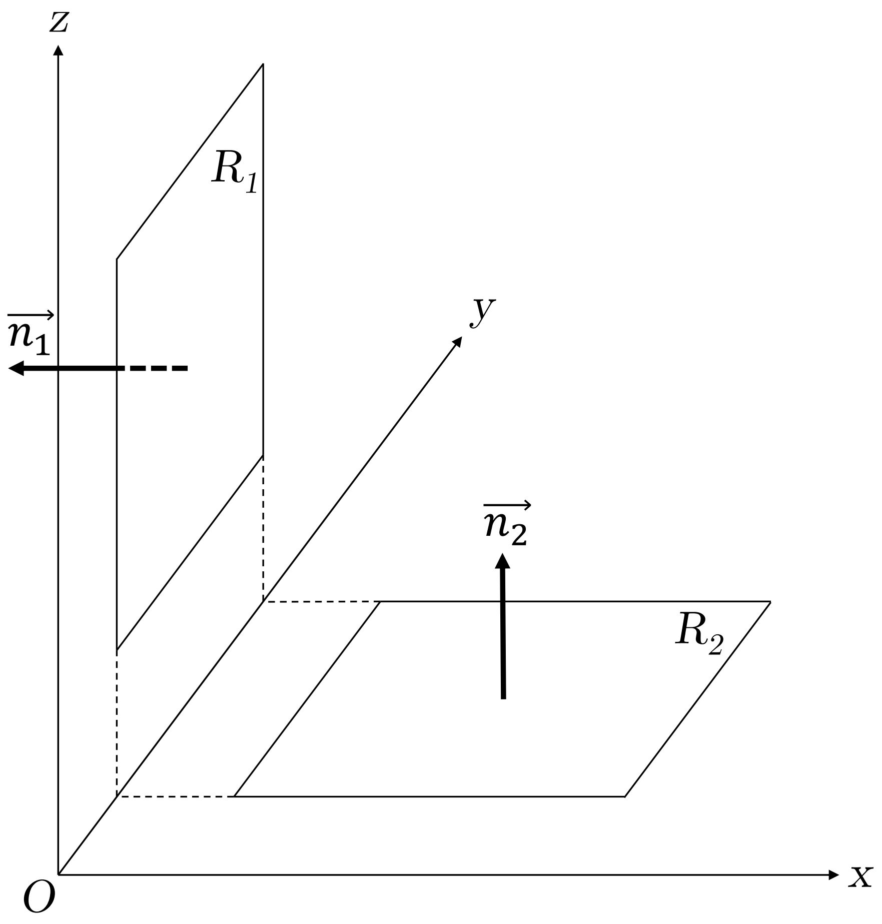


Figure 4.4: An example of two perpendicular (planar) rings R_1, R_2 given normal vectors $\overrightarrow{n_1}$, $\overrightarrow{n_2}$ in three-dimensional space $Oxyz$.

Given an arbitrary plane (representing a ring) $R_1 = (P_1, P_2, \ldots, P_n)$ and a **reference plane** $R_2 = (Q_1, Q_2, \ldots, Q_n), n \geq 3$ in three-dimensional space $Oxyz$ with **unit normal vectors** $\overrightarrow{n_1}$, $\overrightarrow{n_2}$ respectively (see Figure 4.4). The following steps rotate R_1 to a plane R_1' parallel to R_2:

1. Compute the cosine value of $\theta = \angle(\overrightarrow{n_1}, \overrightarrow{n_2})$:

$$\cos(\theta) = \overrightarrow{n_1} \cdot \overrightarrow{n_2} \tag{4.3}$$

   The case of $\cos(\theta) \approx \pm 1$ (given an error tolerance) indicates that both $\overrightarrow{n_1}$ and $\overrightarrow{n_2}$ either point to the same or opposite directions. In other words, R_1 and R_2 are already parallel to one another and thus no further action is required.

2. Find the unit rotation axis $\overrightarrow{u} = (x, y, z)$:

$$\overrightarrow{u} = \frac{\overrightarrow{n_1} \times \overrightarrow{n_2}}{\|\overrightarrow{n_1} \times \overrightarrow{n_2}\|} \tag{4.4}$$

3. Construct the rotation matrix $M_{\overrightarrow{u}}(\theta)$:

$$M_{\overrightarrow{u}}(\theta) = \begin{bmatrix} xxC + c & xyC - zs & xzC + ys \\ yxC + zs & yyC + c & yzC - xs \\ zxC - ys & zyC + xs & zzC + c \end{bmatrix} \tag{4.5}$$

   where:

$$\begin{aligned} c &= \cos(\theta) \\ s &= \sin(\theta) = \sqrt{1 - \cos^2(\theta)} \\ C &= 1 - c \end{aligned}$$

4. Transform all points P_i with $i \in [1, n]$ of R_1 using the constructed rotation matrix:

$$P_i' = M_{\overrightarrow{u}}(\theta) \cdot P_i \tag{4.6}$$

   All transformed points P_i' form the new rotated plane R_1' of R_1, which is now parallel to R_2.

Matching Shapes of Rings

Since two geometrically equivalent rings may still have different orders or numbers of points, their shapes bounded by these points are compared instead. Since the comparison of such shapes are often complex, an open-source third-party geometry library such as the geometry package `java.awt.geom` of the Abstract Window Toolkit (AWT), or the Java Topology Suite (JTS) is applied for the two-dimensional rings rotated previously. The AWT represents rings as `Area`[1] and their points as `Path` objects, where:

- A `Path` describes the outline of an `Area` but may have different point orders or numbers of segments. `Path` objects stored in `Area` are always non-empty and non-overlapping. Empty paths are discarded, while overlapping paths shall be decomposed to separate non-overlapping components;
- `Area` objects are always closed. If an unclosed path or sub-path is given, it shall be automatically enclosed. If such attempt failed, an empty `Area` is returned.

Moreover, some of the most relevant functions offered by this package are:

`boolean contains(double x, double y)`
: Examines whether a point with given coordinates `x` and `y` is located inside the considered `Area`;

`boolean contains(Rectangle2D rect)`
: Examines whether the current `Area` entirely contains the given rectangle;

`boolean equals(Area other)`
: Tests whether the geometries of the two objects are exactly equal;

`void subtract(Area other)`
: Excludes the area of `other` from the interior of current object.

Since the function `equals(Area other)` examines whether the two given geometries are identical, it does not always yield correct results as two geometrically equivalent rings can have different orders or numbers of points. Based on the fact that two geometrically equivalent objects are completely contained in one another, the function `contains(Rectangle2D rect)` can be applied using the minimum bounding rectangle of said objects. However, this also does not yield correct results, as rings are allowed to have shapes different than rectangles as well as two different shapes may have the same minimum bounding rectangle. Alternatively, based on the fact that the subtracted

[1] `http://docs.oracle.com/javase/8/docs/api/java/awt/geom/Area.html`

interior of two geometrically equivalent objects is empty, the function `subtract(Area other)` is applied, where an empty result indicates the geometric equality of given objects. This function generally gives reliable results in most cases but fails to do so in scenarios, where numeric and instrument errors in measured point coordinates persist.

Therefore, specifically taking error tolerance into account, the class `Area` is extended with an additional function called `fuzzy_contains(Area other)` (see Algorithm 7), which returns true if the current `Area` completely contains `other`, otherwise false. The key idea is to also consider all other eight "fuzzy" locations on the boundary of the neighbourhood $N(\epsilon)$ (see Section 4.2.1) of each point along the path of `other`. Line 11 of Algorithm 7 employs the function `contains(double x, double y)` of class `Area` in AWT shown previously. Then:

$$\boxed{\begin{aligned}&\texttt{Ring}\ R_1 \text{ is matched with } R_2\\ &\iff \texttt{Area}\ A_1 \text{ of } R_1 \text{ and } A_2 \text{ of } R_2 \text{ are geometrically equivalent}\\ &\iff A_1.\texttt{fuzzy\_contains}(A_2) \text{ and } A_2.\texttt{fuzzy\_contains}(A_1).\end{aligned}} \tag{4.7}$$

Algorithm 7: `fuzzy_contains(other_area)`

```
Input  : An Area object other_area
Output: true if this completely contains the interior of other_area, else false
1  let path be the set of all points describing the outline of other_area;
2  foreach point p in path do
3      p1 ← $(p.x - \epsilon, p.y)$;
4      p2 ← $(p.x - \epsilon, p.y + \epsilon)$;
5      p3 ← $(p.x, p.y + \epsilon)$;
6      p4 ← $(p.x + \epsilon, p.y + \epsilon)$;
7      p5 ← $(p.x + \epsilon, p.y)$;
8      p6 ← $(p.x + \epsilon, p.y - \epsilon)$;
9      p7 ← $(p.x, p.y - \epsilon)$;
10     p8 ← $(p.x - \epsilon, p.y - \epsilon)$;
11     if this does not contain any of {p, p1, p2, ..., p8} then
12         return false;
13     end
14 end
15 return true;
```

Ring candidates matched by Equation (4.7) require no further comparison.

4.2.4 Matching Geometry of Polygons

Polygons are extensively used in CityGML as a means to describe surfaces of e.g. buildings and building parts. A polygon consists of exactly one exterior ring and an arbitrary number of interior rings, all of which can be represented by `LinearRings` and must be on the same plane. While an exterior ring defines the outline, interior rings define holes in a polygon. Hence, the following rules must hold in polygons [Cox+04; Grö+12; Grö10]:

- Interior rings are completely included inside the polygon, whose outline is bounded by a respective exterior ring;
- No interior ring intersects or contains another in the same polygon;
- Interior rings may touch the exterior ring at a finite number of points as long as the inner area of the respective polygon remains however connected.

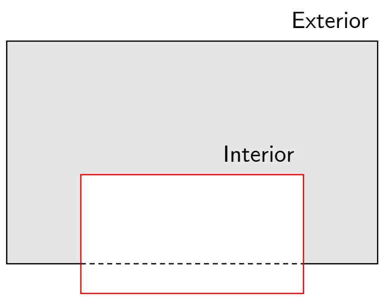
An illustration of the above-mentioned requirements regarding polygon exterior and interior rings can be found in Figure 4.5.

Since exterior and interior rings are basically `LinearRings`, they can be represented by the `Area` class of the AWT. Moreover, as interior rings define holes to be excluded from the polygon's inner area, the function `substract(Area other)` described in Section 4.2.3 can be applied.

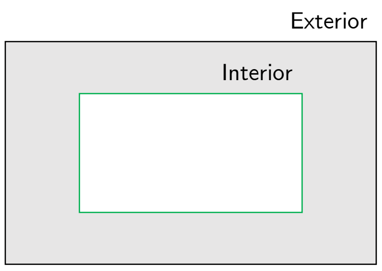
Thus, to compare the geometry of two polygons:

1. Rotate one polygon to a plane parallel to that of the other, or rotate both polygons to a predefined reference plane using Equations (4.3) to (4.6) in Section 4.2.3;
2. Then, for each polygon, create an `Area` object representing its exterior;
3. For each polygon, subtract all its interior areas from the constructed exterior area by calling the function `subtract(Area other)` introduced in Section 4.2.3;
4. The modified `Area` objects then represent the given polygons. To compare their geometries, simply apply Equation (4.7) in Section 4.2.3. Geometrically matched polygons are also considered equal and require no further comparison.

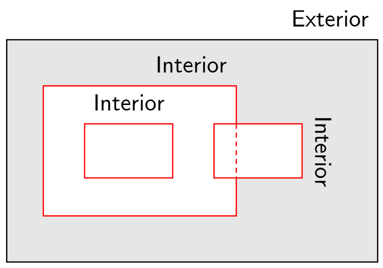
Furthermore, these steps can be extended for the comparison of `Surface`, `OrientableSurface`, `MultiSurface` and `CompositeSurface` objects.



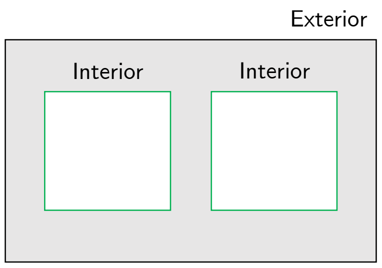
(a) Not a Polygon due to the interior ring (red) not being included in the exterior ring.



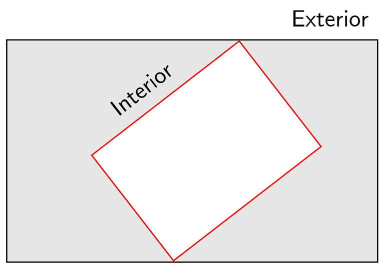
(b) A correct Polygon due to the interior ring (green) being included in the exterior ring.



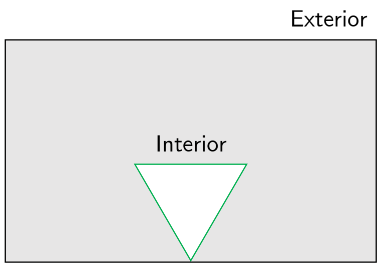
(c) Not a Polygon due to the interior rings (red) overlapping or containing other interior rings.



(d) A correct Polygon due to the interior rings (green) not overlapping one another.



(e) Not a Polygon due to the interior rings (red) dividing the interior of the polygon into two separate parts.



(f) A correct Polygon due to the interior ring (green) only touching the exterior ring at a single point.

Figure 4.5: Some examples illustrating the three requirements needed for interior boundaries of `Polygons`: they are completely enclosed by the exterior ring (Figures 4.5a and 4.5b), do not overlap or contain other interior boundaries (Figures 4.5c and 4.5d) and do not divide inner area while touching the exterior ring at a finite number of points (Figures 4.5e and 4.5f). Adapted from [Grö10].

4.2.5 Matching Geometry of Solids

A solid geometry represents a rigid body and is bounded by a set of polygons.

According to [Cox+04; Grö+12; Grö10], the set $C = \{S_1, S_2, \ldots, S_n\}$, $n \geq 4$ containing boundary polygons S_k of a given solid must fulfil the following requirements:

(i) The intersection of two polygons S_k and S_l of C with $k \neq l$ and $k, l \in [1, n]$ is either empty or an edge shared between `LinearRings` of both respective polygons;

(ii) Each edge $e_k = \overline{P_i^k P_{i+1}^k}$ of a `LinearRing` $R_k = (P_1^k, P_2^k, \ldots, P_{m_k}^k)$, defining a polygon $S_k \in C$ is used exactly once for an edge $e_l = \overline{P_j^l P_{j+1}^l}$ of a `LinearRing` $R_l = (P_1^l, P_2^l, \ldots, P_{m_l}^l)$, defining a polygon $S_l \in C$, with $P_i^k = P_{j+1}^l$ and $P_{i+1}^k = P_j^l$;

(iii) The normal vectors of all polygons $S_k \in C$ point to the outside of the solid;

(iv) All polygons $S_k \in C$ are connected forming the dual graph $G_C = (V_C, E_C)$, where V_C and E_C are the set of nodes and edges respectively. Each node $v \in V_C$ represents exactly one polygon of C and edge $e = (v_k, v_l)$ represents an edge shared between two polygons S_k and S_l of C;

(v) For each vertex P of a `LinearRing` of a polygon in C, the graph $G_P = (V_P, E_P)$ built by only polygons and edges touching P is connected. Each node $v \in V_P$ belongs to a polygon, whose ring contains P. Each edge $e = (v_k, v_l) \in E_P$ represents an edge shared between two polygons S_k and S_l that touch P.

The first and second condition dictate that surfaces of a given solid have no holes, while the fourth and fifth condition allow only connected inner volume of the solid.

A matching candidate of a given solid can be determined by using its footprint (i.e. its ground surface as a polygon in the Oxy plane) or minimum bounding box (see Section 4.2.6). However, unlike previously discussed geometric entities, matched solid candidates may still be unequal due to the fact that different solids can have the same footprint or minimum bounding box. Therefore, found candidates are further compared by successively matching their boundary polygons as described in Section 4.2.4.

4.2.6 Matching Geometry of Minimum Bounding Boxes

In contrast to points, line segments, rings and polygons, the minimum bounding boxes of e.g. buildings and building parts are three-dimensional geometric objects. The minimum bounding box of a building is calculated by all its contained geometries, such as ground, wall and roof surfaces (see Section 3.4). The ability to identify buildings correctly based on their geometrical properties, i.e. minimum bounding boxes, play a critical role in the matching process.

In order to match the geometry of two minimum bounding boxes, several different approaches exist per number of dimensions, for instance:

- Match by their centres (1D);
- Match by their shared footprint (2D);
- Match by their shared volume (3D).

The first approach is simple but tends to yield a large number of false positive results, since representing a three-dimensional entity as a one-dimensional point inevitably leads to information loss and geometrical ambiguities. It is therefore difficult to determine a suitable error tolerance ϵ for matching buildings' centres: it should be small enough to limit the number of found nearest centres, but large enough so that the correct match is obtained in the results.

The second approach compares shared footprints (in 2D) of given minimum bounding boxes and thus yields generally more reliable results. However, it cannot distinguish objects' heights as illustrated in Figure 4.6. Thus, to make full use of all information available in each dimension, this approach is extended with the third dimension. As a result, the third approach is proposed, which matches minimum bounding boxes based on their shared volume.

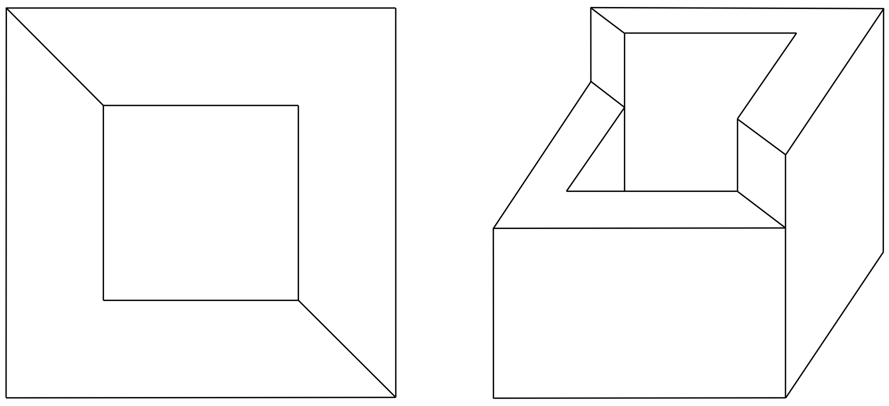


Figure 4.6: An example of two buildings, whose minimum bounding boxes have the same footprint (left) but different heights (right).

Given two arbitrary minimum bounding boxes represented by lower corner points $P = (x_P, y_P, z_P)$, $R = (x_R, y_R, z_R)$ and upper corner points $Q = (x_Q, y_Q, z_Q)$, $S = (x_S, y_S, z_S)$ respectively as illustrated in Figure 4.7, where:

$$\begin{cases} x_P \leq x_Q \wedge y_P \leq y_Q \wedge z_P \leq z_Q \\ x_R \leq x_S \wedge y_R \leq y_S \wedge z_R \leq z_S \end{cases}$$

Their own and shared volume are defined by:

$$V_{PQ} = (x_Q - x_P) \cdot (y_Q - y_P) \cdot (z_Q - z_P) \tag{4.8}$$

$$V_{RS} = (x_S - x_R) \cdot (y_S - y_R) \cdot (z_S - z_R) \tag{4.9}$$

and

$$V_{shared} = d_x \cdot d_y \cdot d_z \tag{4.10}$$

where:

$$\begin{aligned} d_x &= \max\left(\min\left(x_S, x_Q\right) - \max\left(x_R, x_P\right), 0\right) \\ d_y &= \max\left(\min\left(y_S, y_Q\right) - \max\left(y_R, y_P\right), 0\right) \\ d_z &= \max\left(\min\left(z_S, z_Q\right) - \max\left(z_R, z_P\right), 0\right) \end{aligned}$$

The computed shared volume V_{shared} is then compared relatively to the volumes of both minimum bounding boxes:

$$p_{PQ} = \frac{V_{shared}}{V_{PQ}} \tag{4.11}$$

$$p_{RS} = \frac{V_{shared}}{V_{RS}} \tag{4.12}$$

These computed ratios are finally tested against a given threshold $H \in (0, 1]$, which only accepts equal or exceeding values as a match:

$$\begin{aligned} &\text{Minimum bounding boxes } (P, Q) \text{ and } (R, S) \\ &\text{are spatially matched} \\ \Longleftrightarrow &\begin{cases} p_{PQ} & \geq H \\ p_{RS} & \geq H \end{cases} \end{aligned} \tag{4.13}$$

The comparison of both ratios to the predefined threshold H in Equation (4.13) is necessary to exclude scenarios, where a minimum bounding box completely contains the other despite both being not spatially matched. For example, assuming the reference minimum bounding box (P, Q) is completely included in a much larger (R, S). The shared volume V_{shared} is then equal to V_{PQ}. As a result, the ratio p_{PQ} becomes 1. Since $0 < H \leq 1$, the comparison $p_{PQ} \geq 1$ is always satisfied. If this were the only condition, the minimum bounding box (R, S) would be matched with (P, Q), which is not the case in this scenario. To eliminate this, the second comparison $p_{RS} \geq H$ is also examined. With a big enough H, (R, S) is excluded as a result due to its volume being much larger than that of (P, Q) as assumed previously causing p_{RS} to decrease below H.

Therefore, choosing a good and big enough threshold H plays a key role in this step. If H is too high however, no matched minimum bounding box might be found if reasonably large numerical or instrument errors exist. On the other hand, too small H could lead to many false positives. The values of H can be empirically determined based on the error tolerances allowed in the input datasets. Note that a positive match does not always guarantee geometric equality, as different shapes can have the same minimum bounding box.

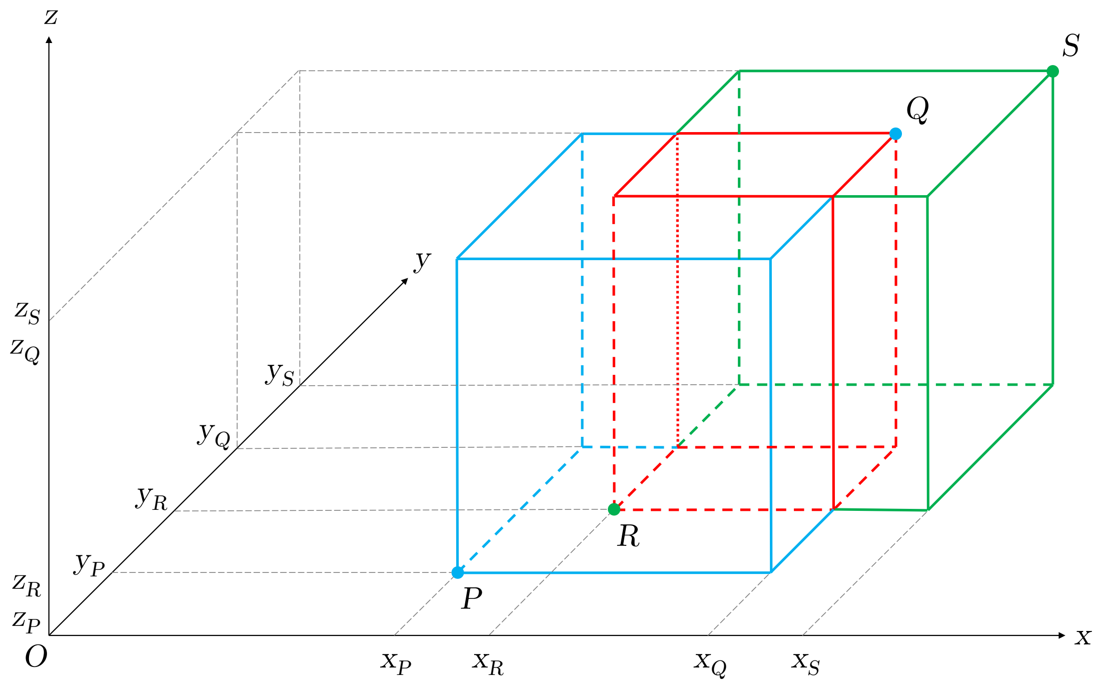


Figure 4.7: An example of the shared volume (red) of two minimum bounding boxes (blue and green), whose lower corner points are P, R and upper corner points are Q, S respectively.

4.3 Spatial Matching Strategies

Section 4.2, particularly Section 4.2.6, determines whether two geometric entities are equivalent and thus can be matched. However, repeatedly matching all available geometric objects is generally computationally expensive, especially if a large number of objects of the same type exist. For instance, the CityGML dataset of Berlin contains approximately 555,000 buildings, comparing the geometry of each possible building pair from each city model results in a quadratic time complexity $\mathcal{O}(n^2)$. Thus, to enable more efficient object retrieval and querying based on their spatial properties, two matching strategies are employed in the course of this research, namely:

- Organizing buildings in a grid layout,
- Organizing buildings in an R-tree.

Both approaches make use of the city models' footprints to assign their buildings in the respective spatial structure, so that the best matching candidates of a given building can efficiently be found. Note that this step serves as the first spatial filtering level, which is fast but may return more than one found candidates (e.g. all buildings contained in the same tile or all buildings whose minimum bounding boxes intersect with a rectangle stored in an R-tree). To narrow these down to one matching candidate, the second filtering level described in Section 4.2 is applied.

4.3.1 Matching in a Grid Layout

By dividing the city model into smaller tiles arranged in a grid layout, it is much more efficient to retrieve a building based on its spatial properties compared to the classic approach, where each possible pair of buildings is successively examined.

Constructing the Grids

To construct a grid for each city model, the following steps are taken:

1. Information such as coordinates of the lower and upper corner are extracted from its envelope or bounding rectangle. If such source is not available, it shall be computed from all of its buildings' minimum bounding boxes as described in Section 3.4. Then, depending on the predefined number of allowed tiles in each dimension, their heights and widths are computed. Alternatively, if tile sizes are given, the total numbers of tiles in each dimension are computed;
2. Furthermore, since two city models are compared using grids, their structures must coincide with one another as illustrated in Figure 4.8. In other words, no

tile from the first grid overlaps with more than one tiles from the second one. Non-coincident grid layouts may prevent the matching process from functioning properly. Therefore, a common point of reference (e.g. the point of origin $O(0,0)$) is selected for both grids. Note that this point must be in the same reference system as that of the two city models as explained in Section 4.2.1;

3. Then, for each city model, a grid is constructed. Moreover, an additional grid boundary made of tiles is created to ensure all buildings are completely contained in the grid and assigned to their respective tile (see Section 4.3.1).

An example explaining these steps can be found in Figure 4.9.

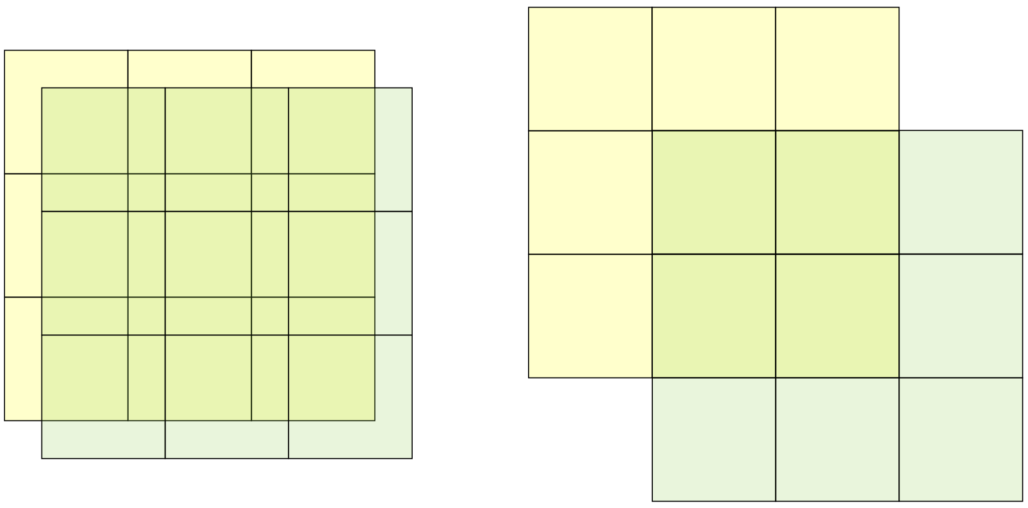

(a) Non-coincident grid layouts. (b) Coincident grid layouts.

Figure 4.8: Examples of non-coincident (Figure 4.8a) and coincident (Figure 4.8b) grid layouts.

Assigning Buildings to Respective Tiles

To assign buildings to the grid constructed from the previous step, their spatial locations relative to the grid are considered, where the footprint of a building may be completely contained in a tile or overlap several tiles simultaneously. In such cases, it is often difficult to determine which tile the building should be assigned to. Incorrectly assigned buildings may lead to incorrect insert and delete operations as the process proceeds. Therefore, to prevent this, a building shall be assigned to all tiles, with which the footprint of its minimum bounding box overlaps.

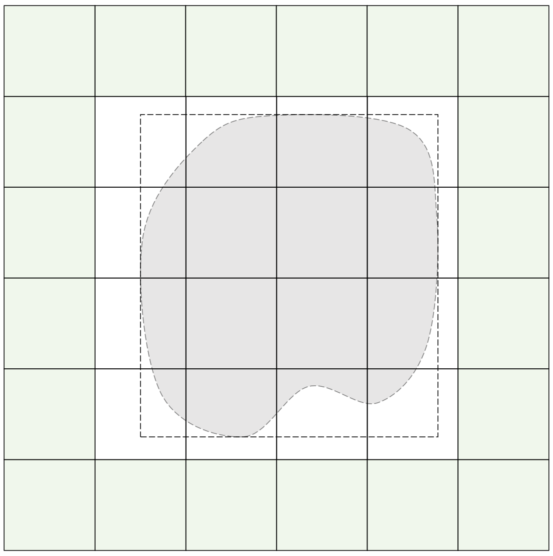


Figure 4.9: An illustration of grid construction. Based on the envelope (dashed) of the footprint (grey) of a given city model, a grid layout (white) is constructed. An additional boundary made of tiles (green) is also created to ensure all buildings assigned to this grid in the next step are included.

To determine whether a building footprint overlaps with a tile, several approaches can be applied by using:

- The footprint's centre,
- The footprint's lower or upper point.

The overlapping test is then performed with the chosen point as described in Algorithm 8.

Explanatory notes:

Line 1 A reference point representing the given building in the grid is initialized. This point can either be the centre or lower or upper corner point of the building footprint as listed previously.

Line 2 A corresponding tile containing the chosen point is found. This can be done by comparing the point's coordinates with those of tile corners in the grid.

Line 3 γ is the allowed distance tolerance near tile borders. In case the distance between the selected point and a tile border is less than or equal to γ, it belongs to both tiles sharing this border.

Algorithm 8: `assign(building, grid)`

```
Input  : A building to be assigned in grid
1  let p be the centre (or lower/upper corner point) of the building's footprint;
2  find tile tile of grid that contains p;
3  let $\gamma$ be the distance tolerance allowed near tile borders;
4  left $\leftarrow$ distance(p, tile.left_border) $\leq \gamma$;
5  right $\leftarrow$ distance(p, tile.right_border) $\leq \gamma$;
6  down $\leftarrow$ distance(p, tile.down_border) $\leq \gamma$;
7  up $\leftarrow$ distance(p, tile.up_border) $\leq \gamma$;
8  tile.add(building);
9  foreach direction of {left, right, down, up, up.left, up.right, down.left, down.right} do
10     if direction then
11         tile.get_neighbor(direction).add(building);
12     end
13 end
```

Lines 4 - 7 The distances between the chosen point p to all four borders of the current tile are computed. If they are within the tolerance range of γ, the respective direction shall be flagged.

Line 8 Assign given building to current tile.

Lines 9 - 13 Depending on flagged directions, the given building may be assigned to more than one tiles contained in the respective 8-neighbourhood.

Note that although Algorithm 8 holds for all positive tile heights and widths, it is recommended to assign them appropriate values. Too small tiles can lead to significant redundancies as a building can be assigned to many tiles simultaneously. Too big tiles may however contain too many buildings and thus slows down the matching process.

Figure 4.10 illustrates how buildings are assigned to tiles accordingly. As shown in Figures 4.10a to 4.10d and 4.10f to 4.10i, a building can be assigned to one or more neighbours of the current tile. Therefore, the additional boundary made of tiles created previously (Figure 4.9) is necessary, as otherwise some buildings may be assigned to non-existing tiles located outside of the city model's footprint.

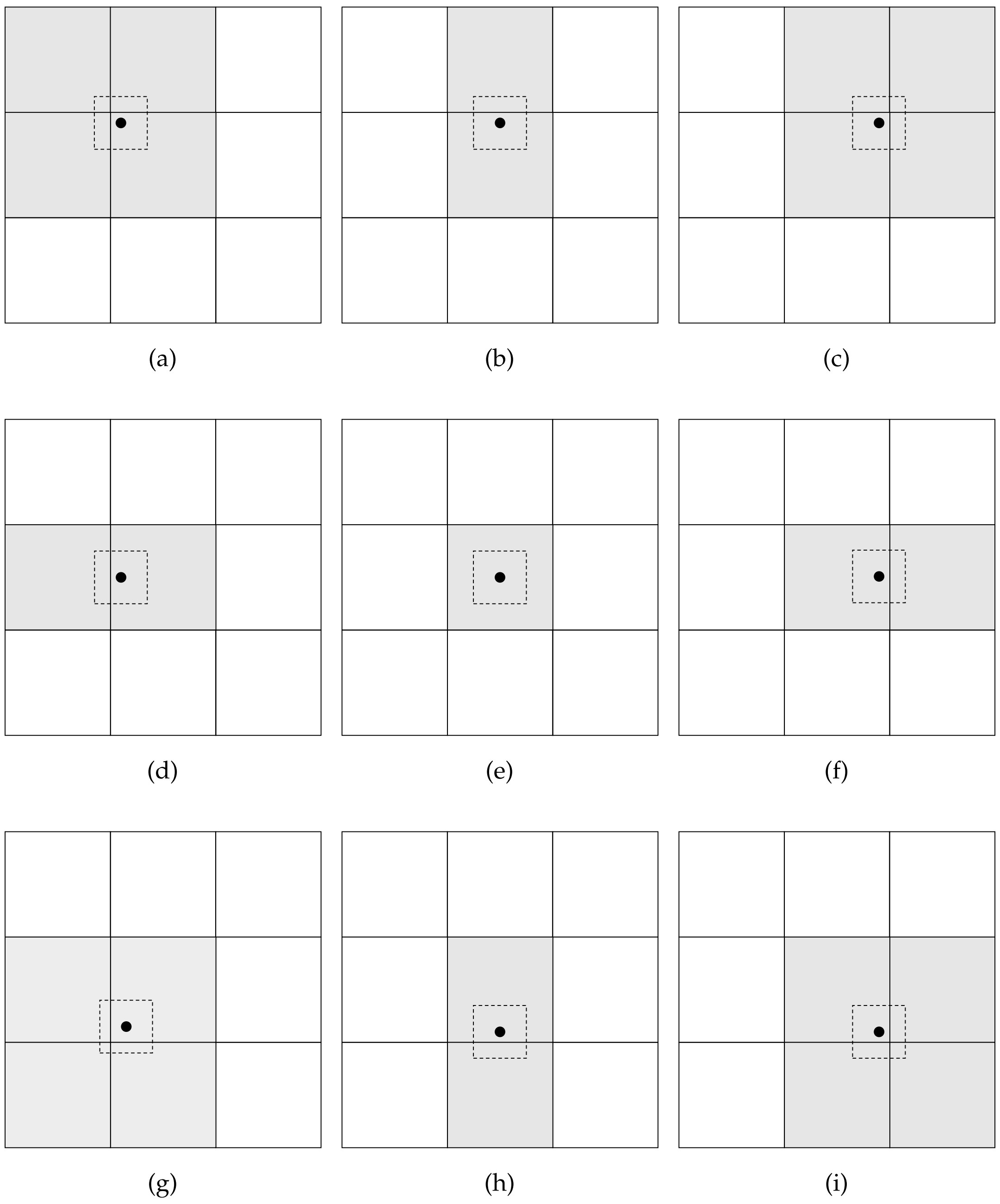


Figure 4.10: An illustration of possible combinations that may occur while assigning buildings (dashed) to tiles in a grid using its centre. Assigned tiles are shown in grey.

Matching Buildings using the Grid Layout

With all preparations completed, the final step is to compare buildings of two city models spatially using their grid layouts as described in Algorithm 9 and illustrated in Figure 4.11.

Explanatory notes:

Line 1 The coincident tiles of both grids are considered.

Lines 2 - 21 Each spatially corresponding pair of tiles in the coincident area from both city models are iterated (Line 2), where:

- **Lines 3 - 14** For each assigned building in one tile (Line 3), its geometric equivalence is searched from the other corresponding tile of the considered pair (Line 4). The function `find_by_geometry` is described in Section 4.2.6. If a spatially equivalent building is found (Line 5), both buildings are removed from all but their current respective tiles (Line 6). Then, they can be matched using the function `match_node` (Line 7) introduced in Algorithm 4. Otherwise, the building is removed from its current respective tile (Line 9) as it does not have a valid match in this tile. If this building do not belong to any tile afterwards (Line 10), a `DELETE` operation shall be created (Line 11).
- **Lines 15 - 20** On the other hand, unmatched buildings in the coincident area of the new city model are removed from their respective tiles (Line 16). If such buildings do not belong to any tile afterwards, an `INSERT` operation shall be attached (Line 18) for each building.

Lines 22 - 29 The remaining non-coincident tiles of the old city model are iterated (Line 22). Assigned buildings (Line 23) are removed from their respective tiles (Line 24). If these buildings do not belong to any tile afterwards (Line 25), a `DELETE` operation shall be created (Line 26).

Lines 30 - 37 Similarly, remaining buildings from the new city model shall be removed from their respective tiles and eventually `INSERT` operations shall be created.

The first advantage of Algorithm 9 is that it makes use of the grid layout to divide city models into smaller tiles and process them locally. With good tile height and width values, the query response time in a grid layout is significantly reduced. Moreover, due to the nature of grids, the approach is compatible with multi-threading without heavy implementation tweaks and adjustments (see Chapter 6). The grid layout's efficiency however depends strongly on its tile sizes and buildings' spatial distribution, which can only be determined empirically. In the worst case scenario, all buildings of a city model might be assigned to a single tile.

Algorithm 9: `match_grid(grid1, grid2)`

```
Input : Grids grid1 and grid2 of old and new city model respectively
let coincident be the coincident area of both grid1 and grid2;
foreach corresponding pair tile1 and tile2 in coincident do
    foreach assigned building1 in tile1 do
        building2 ← tile2.find_by_geometry(building1);
        if building2 is not empty then
            remove building1, building2 from all tiles but tile1, tile2 respectively;
            match_node(building1, building2);
        else
            remove building1 from tile1;
            if building1 does not belong to any tile then
                create a DELETE operation;
            end
        end
    end
    foreach unmatched building2 assigned to tile2 do
        remove building2 from tile2;
        if building2 does not belong to any tile then
            create an INSERT operation;
        end
    end
end
foreach non-coincident tile1 in grid1 do
    foreach assigned building1 in tile1 do
        remove building1 from tile1;
        if building1 does not belong to any tile then
            create a DELETE operation;
        end
    end
end
foreach non-coincident tile2 in grid2 do
    foreach assigned building2 in tile2 do
        remove building2 from tile2;
        if building2 does not belong to any tile then
            create an INSERT operation;
        end
    end
end
```

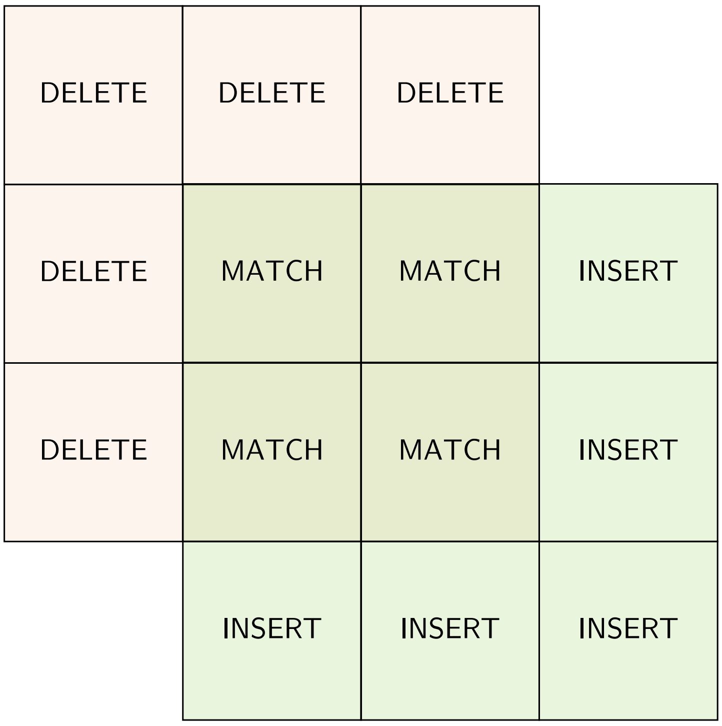


Figure 4.11: An illustration of matching city models by their grid layouts. Buildings in coincident tiles are matched, while those of the remaining tiles of old and new city model are deleted and inserted respectively. Note that a `DELETE` and `INSERT` operation is only executed when the affected building no longer belongs to any tile.

A summary of this strategy's advantages and disadvantages is shown below:

- Advantages:
  - High efficiency and reduced runtime,
  - Compatibility with multi-threading;
- Disadvantages:
  - Strong dependence on the empirically determined tile sizes and buildings' spatial distribution.

4.3.2 Matching in an R-tree

R-trees (as introduced in Section 2.5.1) are an exemplary solution for many problems involving spatial access methods. Since R-trees are balanced, they generally offer a logarithmic time complexity on search operations, which is a significant boost in performance, especially when hundreds of thousands of buildings are to be processed.

Constructing the R-tree

The construction of an R-tree occurs in the plug-in Neo4j Spatial (Section 2.5.2), where R-trees are represented as index layers. Information such as coordinates of lower and upper corner point of city models are not needed beforehand, since an R-tree automatically expands its envelope on the fly. Footprints and minimum bounding boxes of buildings are however required. While iterating, the footprint of each building is extracted or computed if not available (as described in Section 3.4), which then represents its respective building in the R-tree. Splitting and merging nodes in an R-tree are handled by Neo4j Spatial so that the tree structure is always balanced.

Depending on specific implementations, only one R-tree for the newer city model or an R-tree for each city model may be constructed.

Assigning Buildings to the R-tree

The most important task while expanding an R-tree is to link spatial information to data entities it represents, i.e. to link R-tree nodes to their respective buildings. To achieve this, a suitable adapter is needed (Figure 2.14), where a connection between an R-tree node and the minimum bounding box of respective building is established.

Like grid layouts, assigning buildings to an R-tree is executed on the fly while processing their minimum bounding boxes. However, as opposed to grids, a building is assigned to exactly one R-tree node (see Figure 2.14).

Matching Buildings using the R-tree

Algorithm 10 describes the process of matching buildings in two city models using an R-tree.

Explanatory notes:

Line 1 The R-tree layer of the newer city model is considered.

Lines 2 - 10 The main part of the matching process. While iterating over each building in the older city model (Line 2), its matching candidates from the other city model are found by executing the intersection query in the R-tree (Line 3). These

Algorithm 10: `match_rtree(city1, city2)`

```
Input  : City models city1 and city2 to be matched using an R-tree
1  let rtree be the R-tree layer representing city2;
2  foreach building1 in city1 do
3      candidates ← rtree.find_intersection(building1);
4      building2 ← find_by_geometry(candidates);
5      if building2 is not empty then
6          match_node(building1, building2);
7      else
8          create a DELETE operation;
9      end
10 end
11 foreach remaining building2 in city2 do
12     create an INSERT operation;
13 end
```

candidates are then geometrically filtered again so that only the best candidate remains (Line 4). The function `find_by_geometry` is described in Section 4.2.6 and thus the same as in Line 4 from Algorithm 9. If no best candidate is found, a `DELETE` operation shall be created for the reference building (Line 8). Otherwise (Line 5), both buildings are matched (Line 6). The `match_node` function is introduced in Algorithm 4 and hence the same as in Line 7 from Algorithm 9.

Lines 11 - 13 For each remaining unmatched building in the newer city model (Line 11), an `INSERT` operation shall be created (Line 12).

The function `find_intersection` in Line 3 from Algorithm 10 is executed in Neo4j Spatial. The function searches for all rectangles that touch or intersect with given entity, from which linked buildings are retrieved with the help of the constructed adapter. Although R-trees generally offer a logarithmic time complexity in such operations, their efficiency depends greatly on the maximum number of entries M in each internal node (Section 2.5.1). Too large M results in a much shallower tree but increases the number of intersection tests in each internal R-tree node. On the other hand, too small M significantly expands the R-tree's depth and thus increases the number of levels traversed from the root node to each leaf.

To find remaining unmatched buildings in the newer city model in Line 11 of Algorithm 10, an internal list containing all of its buildings can be used. Buildings that have a match shall be removed from this list. Then, remaining elements represent unmatched buildings in the newer city model. The approach is fast but may require a large amount of main memory. Alternatively, an auxiliary node can be created and referenced each time two buildings are matched. Eventually, unreferenced buildings are also unmatched. This does not consume much main memory but may increase the total number of nodes created in the graph database. Finally, the third approach uses an additional R-tree layer for the older city model. Buildings from the newer city model that do not intersect with any rectangles of this R-tree are the remaining unmatched ones. This approach provides fast search operations but may increase the mapping time of the older city model due to the R-tree being expanded on the fly.

The first advantage of using an R-tree is, in particular, the logarithmic time complexity in most query operations. Moreover, with the help of Neo4j Spatial, employing an R-tree while matching is simple and straightforward. In addition, as opposed to the grid layout, each building is assigned to exactly one R-tree node, which reduces reference redundancies and therefore saves runtime spent on matching these redundant objects. However, Neo4j Spatial is not an official plug-in implemented by Neo4j and thus not always compatible with the newest versions of Neo4j. As a result, a downgrade to earlier releases of Neo4j is often required.

A summary of this strategy's advantages and disadvantages is shown below:

- Advantages:
  - High efficiency and performance in logarithmic time complexity,
  - Simplicity and straightforwardness;
- Disadvantages:
  - Incompatibility with newest versions of Neo4j. An older release of Neo4j is therefore required.

5 Updating 3D City Models in CityGML using a Graph Database

The matching process introduced in Chapter 4 creates edit operations on the fly when deviations are detected. No actual data are changed during the process and only edit operations are attached to affected source nodes, so that the updating process in this chapter can execute them accordingly.

5.1 Edit Operations

5.1.1 Class Model

The general model of all edit operations is illustrated in Figure 5.1, where:

- **Abstract class** `EditOperation` is the super class of all edit operations. It defines a `targetNode`, to which the edit operation is attached, and a flag `isOptional` indicating whether said operation should be executed. This flag is mainly used for scenarios where objects are geometrically equivalent but are represented by different syntactic methods (see Section 5.1.2).
- **Abstract class** `EditPropertyOperation` defines all edit operations created on node properties (i.e. object attributes). Thus a `propertyName` is needed to address an affected property. There are three sub-types of edit operations on node properties, namely:
  - `InsertPropertyOperation` defines an `insertValue` of arbitrary type,
  - `UpdatePropertyOperation` defines a `newValue` and `oldValue` of arbitrary type, where the `oldValue` shall be replaced by `newValue`,
  - `DeletePropertyOperation` requires no further attribute.
- **Abstract class** `EditNodeOperation` defines edit operations on the node level (i.e. geo-objects) and has two subclasses, namely:
  - `InsertNodeOperation` defines an `insertNode` reference of type `Node` and an `insertRelType` of `RelationshipType`, both of which are required to uniquely identify the insert position,

– `DeleteNodeOperation` requires no further attribute.

Furthermore, no update operation for nodes is required since it can be realized by a series of `EditPropertyOperation` objects on their properties.

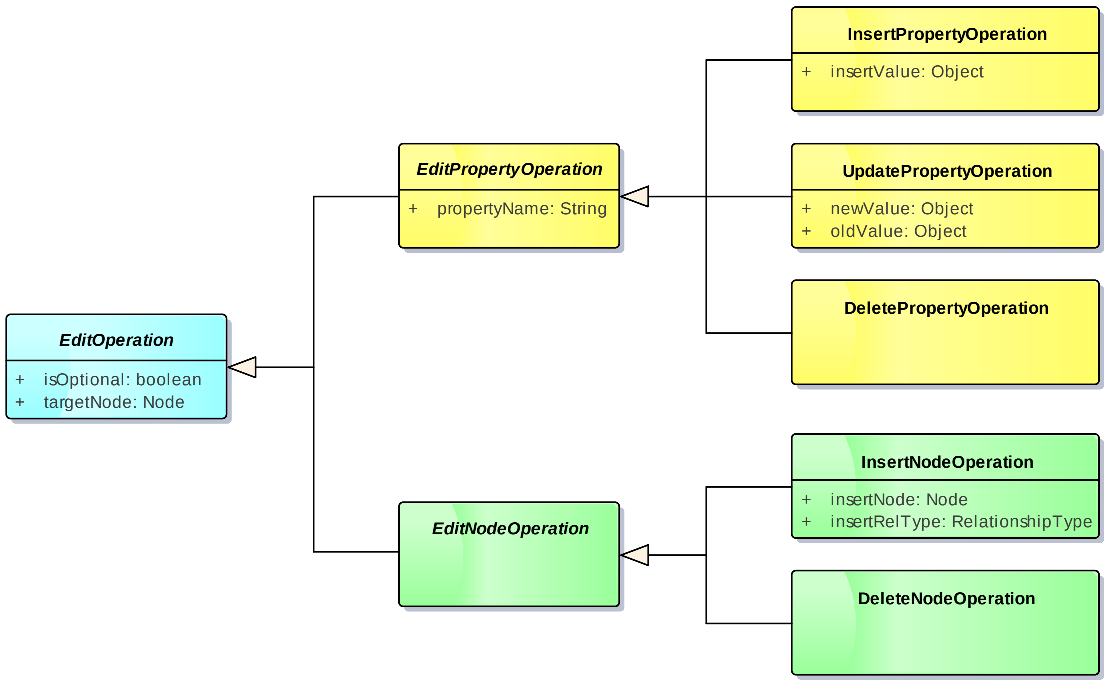


Figure 5.1: A UML class diagram of all edit operations. `EditPropertyOperation` and `EditNodeOperation` objects are shown in yellow and green respectively.

5.1.2 Practical Example

In this section, an example of comparing two CityGML instance documents is demonstrated. Each document contains one building, whose footprint geometry is solely defined by a square-shaped polygon in two-dimensional space Oxy. Both polygons are geometrically equivalent but the first polygon has a single exterior and interior, while the second has one exterior and two interiors (see Figure 5.2). Moreover, these polygons are defined in various syntactic ways allowed in CityGML as shown in Listings 5.1 and 5.2. For instance:

- The interior of the first polygon is bounded by a ring defined by a `posList`, while that of the second consists of a series of individual points `pos`;

- The first polygon contains a hole bounded by a series of `pointProperty` elements, while the two holes of the second polygon are each defined by a `posList`.

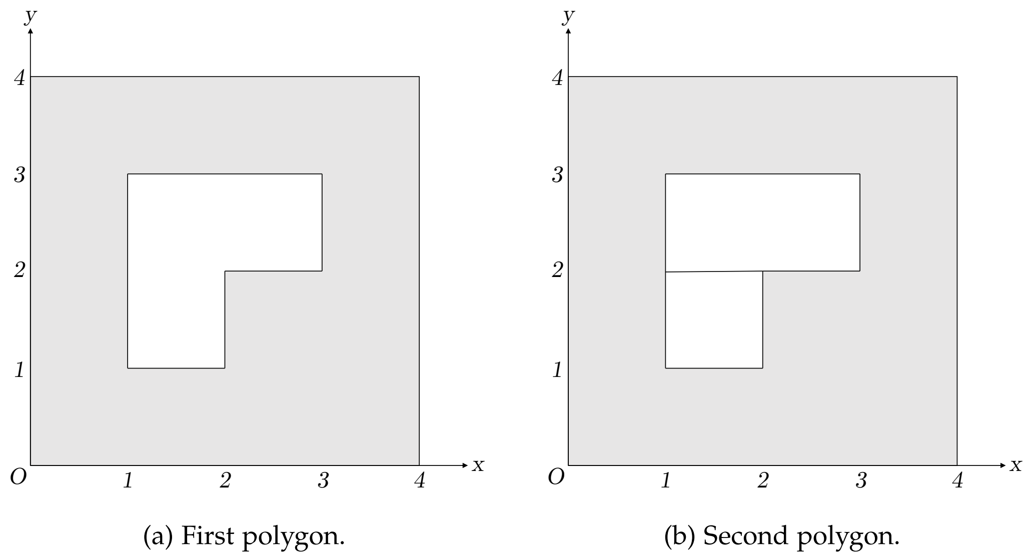


Figure 5.2: An illustration of two geometrically equivalent polygons defined in two CityGML documents. The first polygon (left, Figure 5.2a) consists of an interior (grey) containing a hole (white). The second polygon (right, Figure 5.2b) consists of an interior (grey) with two holes (white).

Listing 5.1: First sample CityGML document.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<CityModel xmlns="http://www.opengis.net/citygml/2.0"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/citygml/building/2.0
  http://schemas.opengis.net/citygml/building/2.0/building.xsd">
  <cityObjectMember>
    <bldg:Building gml:id="BLDG_ID">
      <bldg:boundedBy>
        <bldg:GroundSurface gml:id="GEO_ID">
          <bldg:lod2MultiSurface>
            <gml:MultiSurface>
              <gml:surfaceMember>
```

```
                <gml:Polygon>
                  <gml:exterior>
                    <gml:LinearRing>
                      <gml:posList srsDimension="3">
                        4.0 4.0 0.0
                        0.0 4.0 0.0
                        0.0 0.0 0.0
                        4.0 0.0 0.0
                        4.0 4.0 0.0
                      </gml:posList>
                    </gml:LinearRing>
                  </gml:exterior>
                  <gml:interior>
                    <gml:LinearRing>
                      <gml:pointProperty>
                        <gml:Point>
                          <gml:pos>3.0 3.0 0.0</gml:pos>
                        </gml:Point>
                      </gml:pointProperty>
                      <gml:pointProperty>
                        <gml:Point>
                          <gml:pos>1.0 3.0 0.0</gml:pos>
                        </gml:Point>
                      </gml:pointProperty>
                      <gml:pointProperty>
                        <gml:Point>
                          <gml:pos>1.0 1.0 0.0</gml:pos>
                        </gml:Point>
                      </gml:pointProperty>
                      <gml:pointProperty>
                        <gml:Point>
                          <gml:pos>2.0 1.0 0.0</gml:pos>
                        </gml:Point>
                      </gml:pointProperty>
                      <gml:pointProperty>
                        <gml:Point>
                          <gml:pos>2.0 2.0 0.0</gml:pos>
                        </gml:Point>
                      </gml:pointProperty>
                      <gml:pointProperty>
                        <gml:Point>
                          <gml:pos>3.0 2.0 0.0</gml:pos>
                        </gml:Point>
                      </gml:pointProperty>
                      <gml:pointProperty>
                        <gml:Point>
                          <gml:pos>3.0 3.0 0.0</gml:pos>
                        </gml:Point>
                      </gml:pointProperty>
                    </gml:LinearRing>
                  </gml:interior>
                </gml:Polygon>
              </gml:surfaceMember>
            </gml:MultiSurface>
          </bldg:lod2MultiSurface>
        </bldg:GroundSurface>
      </bldg:boundedBy>
    </bldg:Building>
  </cityObjectMember>
</CityModel>
```

Listing 5.2: Second sample CityGML document.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<CityModel xmlns="http://www.opengis.net/citygml/2.0"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/citygml/building/2.0
  http://schemas.opengis.net/citygml/building/2.0/building.xsd">
  <cityObjectMember>
    <bldg:Building gml:id="BLDG_ID">
      <bldg:boundedBy>
        <bldg:GroundSurface gml:id="GEO_ID">
          <bldg:lod2MultiSurface>
            <gml:MultiSurface>
              <gml:surfaceMember>
                <gml:Polygon>
                  <gml:exterior>
```

```
                          <gml:LinearRing>
                            <gml:pos>4.0 4.0 0.0</gml:pos>
                            <gml:pos>0.0 4.0 0.0</gml:pos>
                            <gml:pos>0.0 0.0 0.0</gml:pos>
                            <gml:pos>4.0 0.0 0.0</gml:pos>
                            <gml:pos>4.0 4.0 0.0</gml:pos>
                          </gml:LinearRing>
                        </gml:exterior>
                        <gml:interior>
                          <gml:LinearRing>
                            <gml:posList srsDimension="3">
                              3.0 3.0 0.0
                              1.0 3.0 0.0
                              1.0 2.0 0.0
                              3.0 2.0 0.0
                              3.0 3.0 0.0
                            </gml:posList>
                          </gml:LinearRing>
                        </gml:interior>
                        <gml:interior>
                          <gml:LinearRing>
                            <gml:posList srsDimension="3">
                              2.0 2.0 0.0
                              1.0 2.0 0.0
                              1.0 1.0 0.0
                              2.0 1.0 0.0
                              2.0 2.0 0.0
                            </gml:posList>
                          </gml:LinearRing>
                        </gml:interior>
                      </gml:Polygon>
                    </gml:surfaceMember>
                  </gml:MultiSurface>
                </bldg:lod2MultiSurface>
              </bldg:GroundSurface>
            </bldg:boundedBy>
          </bldg:Building>
        </cityObjectMember>
</CityModel>
```

After the mapping and matching process are complete, a total number of 15 edit operations are reported, where:

- 8 `DeleteNodeOperation` nodes are created indicating the deletion of 1 `posList` and 7 `pointProperty` objects contained in the exterior and interior of the first document respectively;
- 7 `InsertNodeOperation` nodes are created to insert 5 `pos` and 2 `posList` objects from the exterior and interiors respectively of the second to first city model.

However, all 15 operations are flagged as **optional** indicating that they can be excluded, since both polygons are geometrically equivalent and may remain so. If all these operations are to be executed, the first instance documents shall be updated so that its polygon is identical to that of the second document.

5.2 Updating Building Objects using WFS

The created `EditOperation` nodes can be executed in various ways to update provided CityGML documents. However, large CityGML datasets (e.g. of the city Berlin in the test use case) are often stored in a central database, wherein update operations can be executed over the WFS as introduced in Section 2.6. Thus, this section describes the process of updating a CityGML dataset with the help of created edit operations using the WFS.

Since the WFS operates basically on building features and its properties, the updating process focuses on such objects in given city models. Algorithm 11 describes the controller function of the updating process, where:

Lines 1 - 3 For each building in the older city model (Line 1), the (recursive) function `update` is called (Line 2);

Lines 4 - 6 The remaining buildings in the newer city model that are attached with `InsertNodeOperation` node shall be inserted (Line 5).

The function `update(node)` is described in Algorithm 12, where:

Lines 1 - 4 If the current node is attached with a `DeleteNodeOperation` node, it shall be deleted (Line 2) and the function terminates (Line 3);

Lines 5 - 8 Similarly, if the current node is attached with an `InsertNodeOperation` node, it shall be inserted (Line 6) and the function terminates (Line 7);

Algorithm 11: `update_controller(city1, city2)`

```
Input : Graph representations city1 and city2 of old and new city model resp.
1  foreach building1 in city1 do
2      update(building1);
3  end
4  foreach building2 to be inserted from city2 do
5      insert building2;
6  end
```

Algorithm 12: `update(node)`

```
Input : A node potentially to be updated
1  if node is to be deleted then
2      delete node;
3      return;
4  end
5  if node is to be inserted then
6      insert node;
7      return;
8  end
9  foreach property attached with edit operations in node do
10     if property is to be deleted then
11         delete property;
12     else if property is to be inserted then
13         insert property with new value;
14     else if property is to be updated then
15         update old value of property with new value;
16     end
17 end
18 foreach node child of node do
19     update(child);
20 end
```

Lines Line 9 - Line 17 This part considers all properties contained in the current node (Line 9). In case of an available `DeletePropertyOperation` (Line 10), `InsertPropertyOperation` (Line 12) or `UpdatePropertyOperation` (Line 14) node, actions are taken correspondingly (Lines 11, 13 and 15). Note that, depending on specific implementations, an edit operation may only be executed if it is not optional;

Lines 18 - 20 For each outgoing relationship, i.e. a child of current node (Line 18), the function is called recursively (Line 19).

Both functions make use of the attached `EditOperation` nodes and take actions correspondingly, e.g. delete, insert or update. The next sections explain how such actions can be transformed to WFS requests encoded in HTTP POST (see Section 2.6.1), since the KVP encoding is simply not expressive enough to transport transactions (as shown in Table 2.1). Note that due to the technical limitations of the current version of the WFS as well as of the virtualcityWFS, not all building properties can be updated using these services (such as sub-objects of a building property). Therefore, only properties that can be updated are listed in the following sections.

5.2.1 WFS Transactions on Building Objects

Inserting Buildings

The WFS HTTP POST request shown in Listing 5.3 inserts a new building into the database.

Listing 5.3: WFS insert transaction of buildings.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0">
    <wfs:Insert>
      <bldg:Building gml:id="BLDG_ID">
        <!-- BUILDING_CONTENTS -->
      </bldg:Building>
    </wfs:Insert>
</wfs:Transaction>
```

The `wfs:Insert` element holds the whole contents of the building to be inserted. To retrieve the XML contents of buildings from their respective graph representations, two different approaches can be applied, namely:

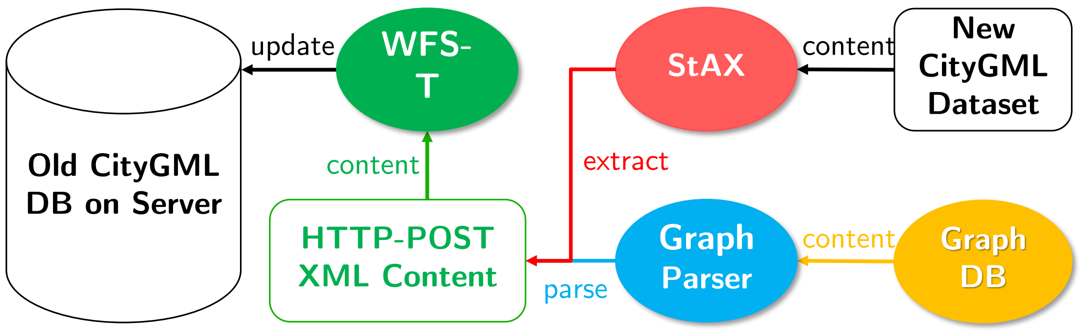


Figure 5.3: Two approaches to retrieving XML contents of a CityGML object: using a Graph-to-CityGML parser (blue) or alternatively, using StAX to extract XML contents directly from source CityGML documents (red).

1. Using a Graph-to-CityGML parser to convert a building node and its respective sub-graphs to 3D city objects encoded in CityGML. In general, the parser can be divided into two steps: graphs are first reversely mapped back to Java objects, which can then be marshalled by using the citygml4j library. The first step is the so-called reverse-mapping process mentioned previously in Section 3.4.1. Thus, this is a multi-purpose solution, as it can be utilized in multiple stages. However, its implementation is estimated to be equally or even more complex than that of the mapping process in Chapter 3, since as opposed to mapping Java objects to graphs, where class hierarchy can be taken advantage of, parsing value-based graphs back to texts encoded in XML is generally more difficult. Due to the time constraints of this thesis, the parser is only partially implemented for geometric objects as described in Chapter 3;

2. Using StAX to extract CityGML elements directly from CityGML documents themselves. This requires CityGML datasets to be available in plain texts as data sources, since it does not make use of the existing graph database. Therefore, this approach is less robust and may be replaced by the Graph-to-CityGML parser in the future.

Figure 5.3 gives an overview of both above-mentioned approaches.

Deleting Buildings

Listing 5.4 describes a delete HTTP POST transaction for building objects, where `BLDG_ID` is the ID of building to be deleted from the database.

Listing 5.4: WFS delete transaction of buildings.

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <wfs:Transaction service="WFS" version="2.0.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0">
  <wfs:Delete typeName="bldg:Building">
    <fes:Filter>
      <fes:ResourceId rid="BLDG_ID"/>
    </fes:Filter>
  </wfs:Delete>
</wfs:Transaction>
```

Updating Buildings

No update or replace transaction for buildings exists in the current version of WFS. However, this can be achieved by deleting and then creating a new building as shown previously.

5.2.2 WFS Transactions on Thematic Properties

The following building attributes can be altered via an OGC compliant WFS 2.0.0:

- Namespace `xmlns:bldg="http://www.opengis.net/citygml/building/2.0"` (by default):
  - `class`
  - `function`
  - `usage`
  - `yearOfConstruction`
  - `yearOfDemolition`
  - `roofType`
  - `measuredHeight`
  - `storeysAboveGround`
  - `storeysBelowGround`
  - `storeyHeightsAboveGround`
  - `storeyHeightsBelowGround`
- Namespace `xmlns:core="http://www.opengis.net/citygml/2.0"` (by default):
  - `creationDate`
  - `terminationDate`
  - `relativeToTerrain`
  - `relativeToWater`

- Namespace `xmlns:gml="http://www.opengis.net/gml"` (by default):
  - `id`
  - `name`
  - `description`

The WFS request shown in Listing 5.5 can be employed to insert, delete or update a building thematic property, where:

`XPath_to_property` An XPath expression (such as `bldg:measuredHeight`) uniquely references the (relative) location of the affected property within its parent building element. To address an attribute of such property, the XPath expression is extended with the attribute name. For instance: `bldg:measuredHeight/@gml:uom` references the attribute `gml:uom` of property `bldg:measuredHeight`. Note that object namespaces must be given correctly. The path `XPath_to_property` must not be empty for all three types of transactions;

`AttributeValue` This value should be empty if the attribute referenced by the `XPath_to_attribute` above is to be deleted. Otherwise, in case the attribute does not exist, the transaction shall insert one with given value. On the other hand, if this attribute does exist, its old value shall be replaced with `AttributeValue`;

`BLDG_ID` The ID of considered building, which must not be empty in all cases.

Listing 5.5: WFS transaction of thematic properties.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0">
  <wfs:Update typeName="bldg:Building">
    <wfs:Property>
      <wfs:ValueReference>XPath_to_property</wfs:ValueReference>
      <wfs:Value>AttributeValue</wfs:Value>
    </wfs:Property>
    <fes:Filter>
      <fes:ResourceId rid="BLDG_ID"/>
    </fes:Filter>
  </wfs:Update>
</wfs:Transaction>
```

5.2.3 WFS Transactions on Geometric Properties

The following building geometric properties can be edited via WFS:

- Namespace `xmlns:bldg="http://www.opengis.net/citygml/building/2.0"` (by default):
  - `lod0FootPrint`
  - `lod0RoofEdge`
  - `lod1Solid`
  - `lod1MultiSurface`
  - `lod1TerrainIntersection`
  - `lod2Solid`
  - `lod2MultiSurface`
  - `lod2MultiCurve`
  - `lod2TerrainIntersection`
  - `lod3Solid`
  - `lod3MultiSurface`
  - `lod3MultiCurve`
  - `lod3TerrainIntersection`
  - `lod4Solid`
  - `lod4MultiSurface`
  - `lod4MultiCurve`
  - `lod4TerrainIntersection`

Inserting and Updating Geometric Properties

Insert and update WFS transactions of geometric properties can be performed by executing the HTTP POST request shown in Listing 5.6, where:

`XPath_to_property` A unique non-empty XPath expression references to the location of affected geometric property. If `XPath_to_property` links to a non-existing object, a corresponding element containing given `XML_CONTENT` shall be created;

`XML_CONTENT` The new non-empty XML value of affected geometric property. If `XPath_to_property` refers to an existing object, its value shall be replaced with `XML_CONTENT`;

`BLDG_ID` The ID of considered building, which must not be empty.

Deleting Geometric Properties

To delete a geometric property from their respective parent building, the attribute `action` of element `wfs:ValueReference` is set to `remove` as shown in Listing 5.7, where both `XPath_to_property` and `BLDG_ID` values must not be empty and the element `wfs:Value` is omitted.

Listing 5.6: WFS insert and update transaction of geometric properties.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0">
  <wfs:Update typeName="bldg:Building">
    <wfs:Property>
      <wfs:ValueReference>XPath_to_property</wfs:ValueReference>
      <wfs:Value>
        <!-- XML_CONTENT -->
      </wfs:Value>
    </wfs:Property>
    <fes:Filter>
      <fes:ResourceId rid="BLDG_ID"/>
    </fes:Filter>
  </wfs:Update>
</wfs:Transaction>
```

Listing 5.7: WFS delete transaction of geometric properties.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0">
  <wfs:Update typeName="bldg:Building">
    <wfs:Property>
      <wfs:ValueReference action="remove">
        XPath_to_property
      </wfs:ValueReference>
    </wfs:Property>
    <fes:Filter>
      <fes:ResourceId rid="BLDG_ID"/>
    </fes:Filter>
  </wfs:Update>
</wfs:Transaction>
```

5.2.4 WFS Transactions on Complex Properties

The ordinary WFS is not expressive enough to support transactional operations on complex XML properties such as external references and CityGML generic attributes as shown in Table 2.2. The current workaround is to delete the entire old building object and insert the new one in the database, which is inefficient. Thus, the vendor-specific extension virtualcityWFS, which provides its own native implementation on inserting, deleting and updating complex elements, is employed.

Inserting Complex Properties

Listing 5.8: WFS insert transaction of generic atributes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
xmlns:vcs="http://www.virtualcitysystems.de/wfs/2.0"
xmlns:gen="http://www.opengis.net/citygml/generics/2.0"
xmlns:gml="http://www.opengis.net/gml"
xmlns:bldg="http://www.opengis.net/citygml/building/2.0"
xmlns:wfs="http://www.opengis.net/wfs/2.0"
xmlns:fes="http://www.opengis.net/fes/2.0"
xmlns:core="http://www.opengis.net/citygml/2.0">
<wfs:Native vendorId="VCS" safeToIgnore="false">
<vcs:InsertComplexProperty typeName="bldg:Building">
<vcs:Property>
<vcs:Value>
<gen:intAttribute name="Gen_name">
<gen:value>Gen_value</gen:value>
</gen:intAttribute>
</vcs:Value>
<vcs:TargetReference>XPath_to_parent</vcs:TargetReference>
</vcs:Property>
<fes:Filter>
<fes:ResourceId rid="BLDG_ID"/>
</fes:Filter>
</vcs:InsertComplexProperty>
</wfs:Native>
</wfs:Transaction>
```

Listing 5.8 describes an insert transaction of generic attributes, where:

`vcs` The namespace `vcs` is a vendor-specific extension of the standard WFS protocols;

`Gen_name`, `Gen_value` Name and value of generic attribute to be inserted respectively;

`XPath_to_parent` The XPath expression to parent element containing this generic attribute. If the parent is a building, element `vcs:TargetReference` can be omitted;

`BLDG_ID` The ID of building object, to which the generic attribute shall be inserted.

Furthermore, the virtualcityWFS also allows the creation of sets of generic attributes as shown in Listing 5.9, where names of the generic set and its elements (`GenSet_name` and `Member_name` respectively) are additionally required.

Listing 5.9: WFS insert transaction of generic atribute sets.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:vcs="http://www.virtualcitysystems.de/wfs/2.0"
  xmlns:gen="http://www.opengis.net/citygml/generics/2.0"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:core="http://www.opengis.net/citygml/2.0">
  <wfs:Native vendorId="VCS" safeToIgnore="false">
    <vcs:InsertComplexProperty typeName="bldg:Building">
      <vcs:Property>
        <vcs:Value>
          <gen:genericAttributeSet name="GenSet_name">
            <gen:stringAttribute name="Member_name">
              <gen:value>Gen_value</gen:value>
            </gen:stringAttribute>
          </gen:genericAttributeSet>
        </vcs:Value>
        <vcs:TargetReference>XPath_to_parent</vcs:TargetReference>
      </vcs:Property>
      <fes:Filter>
        <fes:ResourceId rid="BLDG_ID"/>
      </fes:Filter>
    </vcs:InsertComplexProperty>
  </wfs:Native>
</wfs:Transaction>
```

The external reference objects can be inserted by executing requests shown in Listing 5.10, where the URI to information system `Uri_to_information_system` and external object's name `Name_of_external_object` are required.

Listing 5.10: WFS insert transaction of external references.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:vcs="http://www.virtualcitysystems.de/wfs/2.0"
  xmlns:gen="http://www.opengis.net/citygml/generics/2.0"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:core="http://www.opengis.net/citygml/2.0">
  <wfs:Native vendorId="VCS" safeToIgnore="false">
    <vcs:InsertComplexProperty typeName="bldg:Building">
      <vcs:Property>
        <vcs:Value>
          <vcs:externalReference>
            <core:informationSystem>
              Uri_to_information_system
            </core:informationSystem>
            <core:externalObject>
              <core:name>Name_of_external_object</core:name>
            </core:externalObject>
          </vcs:externalReference>
        </vcs:Value>
      </vcs:Property>
      <fes:Filter>
        <fes:ResourceId rid="BLDG_ID"/>
      </fes:Filter>
    </vcs:InsertComplexProperty>
  </wfs:Native>
</wfs:Transaction>
```

Deleting Complex Properties

Deleting complex properties of a building is similar to that of geometric properties. For instance, to delete a generic attribute, the attribute `action` of `wfs:ValueReference` is set to `remove` and the XPath expression is pointed to target element, e.g. a request with `gen:intAttribute[@gen:name='Gen_name']` shall delete the generic attribute named `Gen_name` in the respective building (see Listing 5.11). Moreover, to delete an external reference element, the XPath expression `wfs:ValueReference` shall simply have the value `core:externalReference`.

Listing 5.11: WFS delete transaction of generic attributes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:vcs="http://www.virtualcitysystems.de/wfs/2.0"
  xmlns:gen="http://www.opengis.net/citygml/generics/2.0"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:core="http://www.opengis.net/citygml/2.0">
  <wfs:Update typeName="bldg:Building">
    <wfs:Property>
      <wfs:ValueReference action="remove">
        gen:intAttribute[@gen:name='Gen_name']
      </wfs:ValueReference>
    </wfs:Property>
    <fes:Filter>
      <fes:ResourceId rid="BLDG_ID"/>
    </fes:Filter>
  </wfs:Update>
</wfs:Transaction>
```

Updating Complex Properties

Complex properties can be updated in the same manner as in geometric properties. Updating the value of a generic attribute requires an XPath expression referencing its (relative) location within the respective parent building. For example, an update operation containing the path `gen:intAttribute[@gen:name='Gen_name']/gen:value` shall replace the old value of a generic integer attribute named `Gen_name` with a new one (see Listing 5.12). The same applies for other types of generic attributes. To replace the whole contents of a generic set or an external reference object, the `wfs:Value` element may contain their body encoded in XML.

Listing 5.12: WFS update transaction of generic attributes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wfs:Transaction service="WFS" version="2.0.0"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:bldg="http://www.opengis.net/citygml/building/2.0">
  <wfs:Update typeName="bldg:Building">
    <wfs:Property>
      <wfs:ValueReference>
        gen:intAttribute[@gen:name='Gen_name']/gen:value
      </wfs:ValueReference>
      <wfs:Value>42</wfs:Value>
    </wfs:Property>
    <fes:Filter>
      <fes:ResourceId rid="BLDG_ID"/>
    </fes:Filter>
  </wfs:Update>
</wfs:Transaction>
```

6 Performance Optimization

This chapter provides an overview of hardware and system configurations as well as adjustments in the implementation for optimal throughput.

6.1 Memory Tuning and Storage Selections

6.1.1 Memory Tuning

Correctly configured and sufficient main memory ensures a stable testing environment and optimal outputs. Three relevant types of main memory are: Operating System (OS) Memory, Page Cache and Heap Space, whose total sum must not exceed the total amount of available main memory in the testing system as described in Equation (6.1).

$$\boxed{\text{OS Memory} + \text{Page Cache} + \text{Heap Space} \leq \text{Available RAM.}} \tag{6.1}$$

OS Memory

Although the availability of a Neo4j instance has the highest priority, the hosting OS should be provided with enough main memory to handle OS-specific processes. In addition, since indexing in Neo4j requires a buffer cache that loads `graph.db/index` and `graph.db/schema` directory into main memory, the OS memory should also include this amount as shown in Equation (6.2) [Neo17d].

$$\boxed{\begin{aligned}\text{OS Memory} = {} & \text{(Reserved memory for non-Neo4j processes)}\\ & + (\text{Size of } \texttt{graph.db/index}) + (\text{Size of } \texttt{graph.db/schema}).\end{aligned}} \tag{6.2}$$

Page Cache

Most of Neo4j database is stored on disk but can be loaded back into main memory when needed. To avoid costly disk operations, a page cache as a reserved portion of available RAM is used. To determine the needed main memory for the page cache, [Neo17d] proposes to add up all sizes of `graph.db/*store.db*` files and leave an additional 20% of that amount for potential growth as described in Equation (6.3).

$$\boxed{\text{Page Cache} = 1.2 \cdot \sum \left(\text{Size of } \texttt{graph.db/*store.db*}\right).} \tag{6.3}$$

To determine the sizes of `graph.db/*store.db*` files for a new database, it is recommended to first perform a test import of n-th the size of the actual data and investigate their sizes. The expected sum can then be estimated by multiplying this with n.

Heap Space

Both Neo4j and the Java Virtual Machine (JVM) share the same heap space, where all internal variables are temporarily stored. [Neo17d] suggests a minimum amount of 8 or 16 GB of available RAM should be reserved for the heap space. However, for large input datasets as in the test use case, a larger heap space might be needed.

Garbage Collector

The Garbage Collector (GC) is a complex background memory management process in the JVM that automatically reclaims objects occupied in main memory that are no longer in use. A generational GC divides the heap space into at least two groups: young generation and old generation. Newly created objects are allocated in the young generation and shall later be moved to the old generation if its lifespan exceeds a specific period of time. Each time a generation fills up, the GC pauses all running threads and performs a collection. This pause time in the young generation correlates with the live set of objects and is thus generally faster than that in the old generation, where pause time roughly correlates with the whole heap's size [Neo17d]. Therefore, ideally, database transaction and query objects should never be moved to the old generation.

To achieve this, the size ratio between the new and old generation can be configured by the flag `-XX:NewRatio`. For instance,

`-XX:NewRatio=`n

indicates that the old generation is n times as large as the young generation. Typical values of n range between 2 and 8. If the implementation changes a large number of database objects, then the ratio n should be small (e.g. $n = 1$) to increase the size of the young generation. However, too big young generation forces the GC to collect objects in the filled up old generation more frequently, which performs a collection in the whole heap space as explained previously. On the other hand, too small young generation causes the so-called "premature promotion", where objects are moved to the old generation too early due to the young generation running out of space.

Furthermore, the initial and maximum heap space can be adjusted with the help of the flags `-Xms` and `-Xmx` respectively. A heap space starts with the initial size and will be expanded towards the maximum value once the current heap space has no available free space. Each expansion triggers a call on the GC. Therefore, to prevent the costly collection for every heap growth, [Neo17d] recommends that both initial and maximum size of heap space should be set to the same value. For example,

```
-Xms16384m -Xmx16384m
```

assigns the initial and maximum size values of the heap space to 16 GB. For the large CityGML datasets of Berlin tested in this research, an initial heap space of at least 12 GB is recommended.

Note that to save main memory, the JVM employs the Compressed Ordinary Object Pointer (OOP) feature that compresses object references. The feature can be enabled by using the flag

```
-XX:+UseCompressedOops
```

in `JDK` 6 before the 6u23 release. In newer versions, the feature is enabled by default. In Java SE 7, the compressed OOPs is activated in 64-bit JVM if the value of flag `-Xmx` is undefined or smaller than 32 GB [Ora16]. Therefore, a maximum heap space size value of 32 GB and beyond shall deactivate the compressed OOP and thus might cause marginal or negative gains in performance, unless the increase in size is significant (64 GB or above) as stated in [Neo17d].

Finally, since some long-lived objects in Neo4j may remain in the old generation, the use of a concurrent GC is suggested:

```
-XX:+UseG1GC.
```

6.1.2 Storage Selections

A storage of sufficient free capacity is required to store the Neo4j database. For instance, mapping and matching two city models of the city Berlin with approximately 550,000 buildings per model requires a storage allocation of at least 200 GB (incl. transaction logs).

Moreover, since Neo4j communicates back and forth with storage devices, its performance depends greatly on their read and write speed. For instance, an SSD is noticeably faster in this regard than an HDD. However, both are much less responsive than a RAM-disk, which is a partition mounted directly from the system's main memory. The RAM-disk is however volatile, meaning that all stored data shall be lost once the system reboots.

6.2 Batch Transaction Processing

As described in Section 2.4.5, all database operations must be declared inside a transaction. A transaction holds affected resources in main memory and locks them until it is closed. In case of a successful transaction, its effects shall be committed to the database storage, which is generally slow due to costly disk read and write operations. Therefore, database operations are often performed and committed together in batches. For instance, to create 1000 nodes, instead of creating 1000 transactions, one transaction for each node, thus requiring 1000 costly disk write operations, the same process can be achieved by using only one transaction, which can reduce the runtime significantly.

The number of database operations per transaction (or batch size) is often empirically determined. Too small batch size results in minimal gains in performance, while too many database operations in a batch increases the demand on main memory as well as the frequency of GC calls.

6.3 Concurrent Processing

Database performance can be improved significantly using concurrency. This section introduces some available multi-threading approaches and then briefly describes the used deadlock avoidance mechanism.

6.3.1 Multi-threading Approaches

Classic Approach

Typically, to achieve concurrency, each database operation is submitted to a new thread. Threads are then executed simultaneously. This classic approach is simple to implement but may provide marginal or negative gains on performance due to the following problems:

- If the majority of submitted workloads consist of of only one or a few database operations each, then runtime is spent mostly on switching contexts between threads;
- Since transactions are bound to their respective thread in Neo4j, a transaction must be closed before its thread ends. This means that too many transactions for too few operations per thread may result in an excessively large number of costly disk writes;
- Since a transaction does not exist outside of its thread, processing database operations in batches is difficult without further adjustments.

Producer-consumer Approach

The producer-consumer approach is one of most well-known solutions for the multi-process synchronization problems. This design pattern consists of two types of processes: producer(s) and consumer(s), where at least one producer and one consumer must exist. The producers produce data simultaneously, which are then placed into a buffer shared among all producers and consumers. At the same time, the consumers attempt to remove the data from the buffer. If none is available, the consumers wait until the next cycle. Otherwise, retrieved data are consumed and removed from the buffer. Consumers also consume data simultaneously (see Figure 6.1).

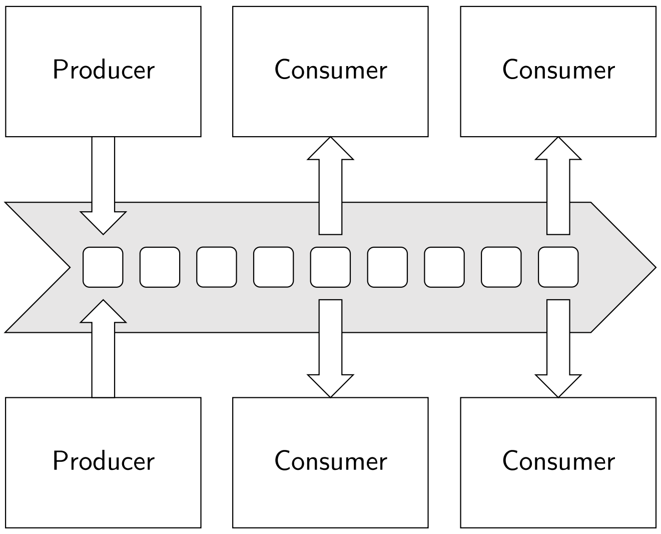


Figure 6.1: An example of two producers and four consumers in the Producer-consumer design pattern.

In Java, producers and consumers can be represented as threads, while the buffer shared among them is realized as a blocking queue [Goe+06]. The advantages of this approach are listed as follows:

- This design pattern is particularly suitable to the mapping process, where a producer unmarshals CityGML elements to Java objects and puts them into the queue, while the consumers transform these to respective graph representations;
- The blocking queue provides a separate view between the producers and consumers, and even among producers and consumers. In other words, producers and consumers function independently from each other;

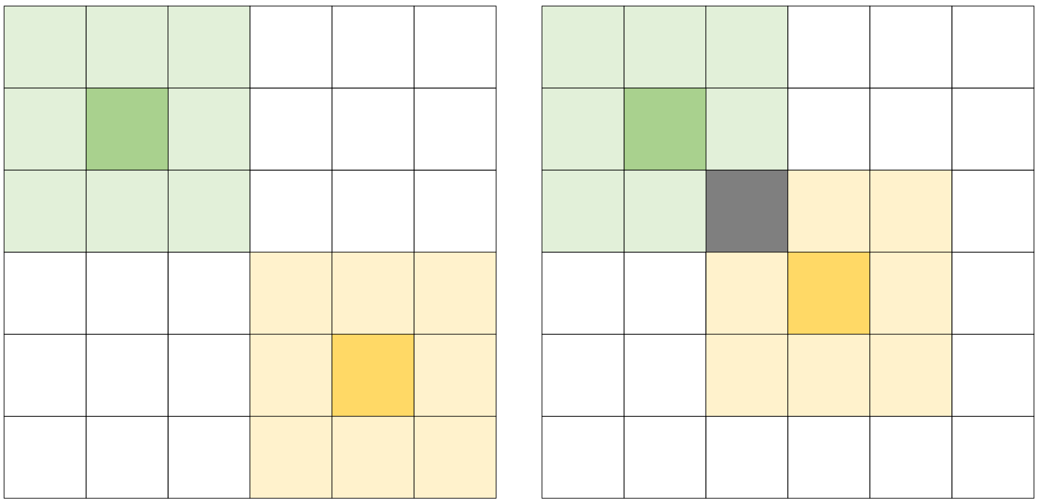
- Executing database operations in batches is possible in this approach, since each producer and consumer is a thread. For instance, the consumers remove *n* Java feature objects from the queue, map them to graphs and then commit these changes together to the database.

The total number of producers and consumers N are often determined by the maximum number of threads that can be handled simultaneously by the system processor. The ratio between the number of producers N_p and consumers N_c depends however on specific use cases. Too many producers with an unbounded buffer often quickly leads to insufficient memory error, since there are not enough consumers to consume produced data, especially if the workloads of the consumers are more computationally intensive. On the other hand, if consuming is not as intensive as producing data, a combination of several producers and consumers is possible.

6.3.2 Deadlock Avoidance

Parallel processing offers great gains on performance but often comes with deadlocks when threads concurrently attempt to write a shared object and block one another. For example, while matching buildings from two city models using grids as explained in Section 4.3.1, since a building may be assigned to multiple tiles in its neighbourhood, potential deadlocks are detected if concurrent threads attempt to access and modify buildings assigned to the same neighbouring tiles (see Figure 6.2).

To avoid deadlocks, objects are isolated from each other so that threads are conflict-free and do not (write-) block each other at runtime. For instance, in the matching process using a grid layout, threads are executed in such an order that no two buildings with overlapping neighbourhoods are processed simultaneously (as shown in Figure 6.2a). Moreover, database transactions should be closed as soon as possible so that they do not block too many objects for a prolonged period of time.



(a) No deadlock is detected. (b) A potential deadlock is detected.

Figure 6.2: An illustration of a deadlock encountered while concurrently matching buildings using a grid layout. Figure 6.2a shows two tiles (dark green, dark orange) containing buildings to be matched and their respective neighbouring tiles (light green, light orange) with no deadlock detected. On the other hand, a potential deadlock is detected in Figure 6.2b since a tile (grey) is shared between threads at runtime.

7 Application Results and Discussion

The previously proposed theoretical approaches are applied to real-world input data in this chapter, where several experiments with various test configurations are performed. The results recorded during these experiments are then presented and discussed.

7.1 Test Setup

7.1.1 Testing Environment

The majority of experiments in this chapter are performed on a dedicated server-class machine provided with the following specifications:

- Operating system: SUSE Linux Enterprise Server 12 SP1 (64 bit),
- CPU: 2x Intel® Xeon® E5-2667 v3 @3.20GHz (16 CPUs in total + Hyper-threading),
- Storage: a Solid-state Drive Array (SSD) connected via PCIe,
- RAM: 1 TB in total.

7.1.2 Input Data

The real-world input data applied in this chapter is the entire 3D city model of Berlin encoded in CityGML v2.0.0. It contains LOD2 information of 539,182 buildings within the city area of 890 km^2 and occupies 15.5 GB in physical storage.[1] The new dataset contains changes (such as replacing in-line objects with XLinks and vice versa, adding small deviations in points' coordinates, using different equivalent geometries for polygon boundaries, etc.) made manually to the old one.

7.1.3 Test Configurations

Java Virtual Machine (JVM) Configurations

Both the testing system and Neo4j share the same JVM, which is provided with the following run configurations:

[1]The datasets are available under `http://www.businesslocationcenter.de/en/downloadportal`.

- Initial and maximum heap space size: 30,000 MB (`-Xms30000m` and `-Xmx30000m`),
- Garbage collector: concurrent `G1GC` (`-XX:+UseG1GC`).

Program Configurations

The application can be executed using various configurations, some of which are:

- The choice between single and multi-threading in the mapping and matching process. In case of multi-threading (Section 6.3.1), the number of producers as well as consumers per producer are adjustable;
- XLinks can be resolved using either Neo4j's (built-in legacy or manual) indices stored on disk or internal hash maps held in main memory (Section 3.3.2);
- Both of the mapping and matching process can be executed using different numbers of database operations as well as features and buildings wrapped per batch transaction (Section 6.2);
- It is possible to employ different spatial search strategies, namely the grid layout and R-tree, in the matching process. In case of a chosen grid layout (Section 4.3.1), the unit tile sizes (i.e. height and width) are editable. Otherwise, if an R-tree is applied (Section 4.3.2), the maximum entries M per internal node can be adjusted.

The application configurations are empirically determined to ensure a stable testing environment and optimum throughput. Unless otherwise specified, the default configurations of all experiments performed in the next sections are as follows:

- Mapping:
  - Multi-threading: enabled with 1 producer and 15 consumers,
  - Index storage: internal hash maps held in main memory,
  - Split CityGML documents: per collection member (top-level feature),
  - Building batch size per one transaction: 10,
  - Database operation batch size per one transaction: 5000;
- Matching:
  - Multi-threading: enabled with 1 producer and 15 consumers,
  - Spatial search strategy: R-tree,
  - Maximum number of entries M per internal R-tree node: 10,
  - Building batch size per one transaction: 10,
  - Database operation batch size per one transaction: 5000.

7.2 Application Results

7.2.1 Statistics of Mapped Graph Database

Table 7.1 shows node labels and their respective frequencies (in descending order) in the graph database after the mapping process of two CityGML datasets of Berlin is complete. The total number of nodes created by the mapping process is 321,142,046. Note that unlabelled and auxiliary nodes such as R-tree nodes are not listed in the table.

Node label	No. nodes	Percentage
SURFACE_DATA_PROPERTY	30,940,736	9.635%
STRING_ATTRIBUTE	23,941,698	7.455%
COLOR	17,902,720	5.575%
SURFACE_PROPERTY	15,407,528	4.798%
CODE	14,950,860	4.656%
DIRECT_POSITION	13,887,228	4.324%
DIRECT_POSITION_LIST	12,939,316	4.029%
LINEAR_RING	12,939,316	4.029%
EXTERIOR	12,928,580	4.025%
POLYGON	12,928,580	4.025%
TEXTURE_COORDINATES	11,532,330	3.591%
TEX_COORD_LIST	11,526,400	3.589%
_TEXTURED_SURFACE	11,526,400	3.589%
COLOR_PLUS_OPACITY	11,520,154	3.587%
PARAMETERIZED_TEXTURE	11,520,154	3.587%
TEXTURE_TYPE	11,520,154	3.587%
WRAP_MODE	11,520,154	3.587%
BOUNDING_SHAPE	6,943,614	2.162%
ENVELOPE	6,943,614	2.162%
MULTI_SURFACE	5,864,808	1.826%
MULTI_SURFACE_PROPERTY	5,864,808	1.826%
BUILDING_BOUNDARY_SURFACE_PROPERTY	5,864,792	1.826%
INT_ATTRIBUTE	4,041,104	1.257%
DOUBLE_ATTRIBUTE	3,510,252	1.092%
BUILDING_WALL_SURFACE	3,245,878	1.011%
APPEARANCE	3,235,024	1.007%
APPEARANCE_PROPERTY	3,235,024	1.007%
X3D_MATERIAL	2,127,522	0.662%
BUILDING_ROOF_SURFACE	1,524,522	0.475%

Node label	No. nodes	Percentage
ADDRESS	1,172,086	0.365%
ADDRESS_DETAILS	1,172,086	0.365%
ADDRESS_PROPERTY	1,172,086	0.365%
COUNTRY	1,172,086	0.365%
LOCALITY	1,172,086	0.365%
LOCALITY_NAME	1,172,086	0.365%
XAL_ADDRESS_PROPERTY	1,172,086	0.365%
BUILDING_GROUND_SURFACE	1,094,392	0.341%
EXTERNAL_OBJECT	1,078,818	0.336%
EXTERNAL_REFERENCE	1,078,818	0.336%
BUILDING	1,078,364	0.336%
CITY_OBJECT_MEMBER	1,078,364	0.336%
THOROUGHFARE	1,028,160	0.320%
THOROUGHFARE_NAME	1,028,160	0.320%
THOROUGHFARE_NUMBER	1,028,160	0.320%
THOROUGHFARE_NUMBER_OR_RANGE	1,028,160	0.320%
LENGTH	978,336	0.305%
COMPOSITE_SURFACE	149,692	0.047%
SOLID	149,570	0.047%
SOLID_PROPERTY	149,570	0.047%
COUNTRY_NAME	143,800	0.045%
INTERIOR	10,736	0.003%
BUILDING_PART	458	0.001%
BUILDING_PART_PROPERTY	458	0.001%
STRING_OR_REF	152	0.001%
URI_ATTRIBUTE	4	0.001%
CITY_MODEL	2	0.001%
Total	321,142,046	100.000%

Table 7.1: Labelled nodes and their respective frequencies after two CityGML instances of the whole city Berlin are mapped.

The graph database allocates a total number of 132,158,372 bytes (or approximately 126 GB) of disk storage in total. However, this does not include the input test data as well as the `graph.db/index` directory used for indexing in Neo4j, which could additionally add up about several to tens of gigabytes to the total allocation.

7.2.2 Single and Multi-threading Performance

The differences in performance between the single-threaded and some combinations of different numbers of producers and consumers in the multi-threaded processing are shown in Figure 7.1. Result discussion and explanation can be found in Section 7.3.

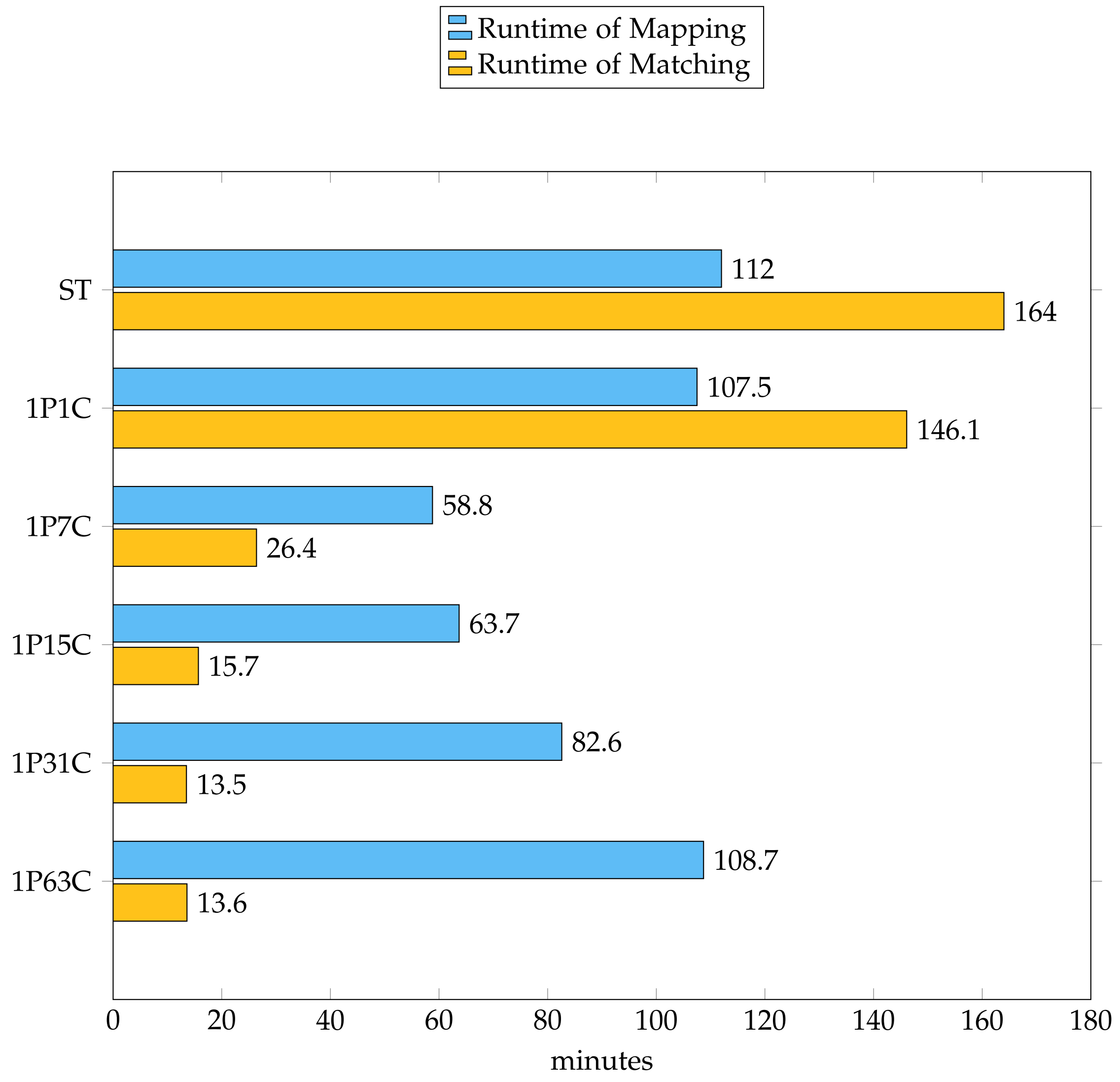


Figure 7.1: The differences in performance between the single-threaded (ST) and some combinations of different numbers of producers and consumers in the multi-threaded processing, where *m*P*n*C denotes *m* producer(s) and *n* consumer(s).

7.2.3 Indexing Performance

The impact of storing indices on disk (Neo4j's built-in indices) and in main memory (self-developed indexing mechanism using internal hash maps) on performance is shown in Figure 7.2.

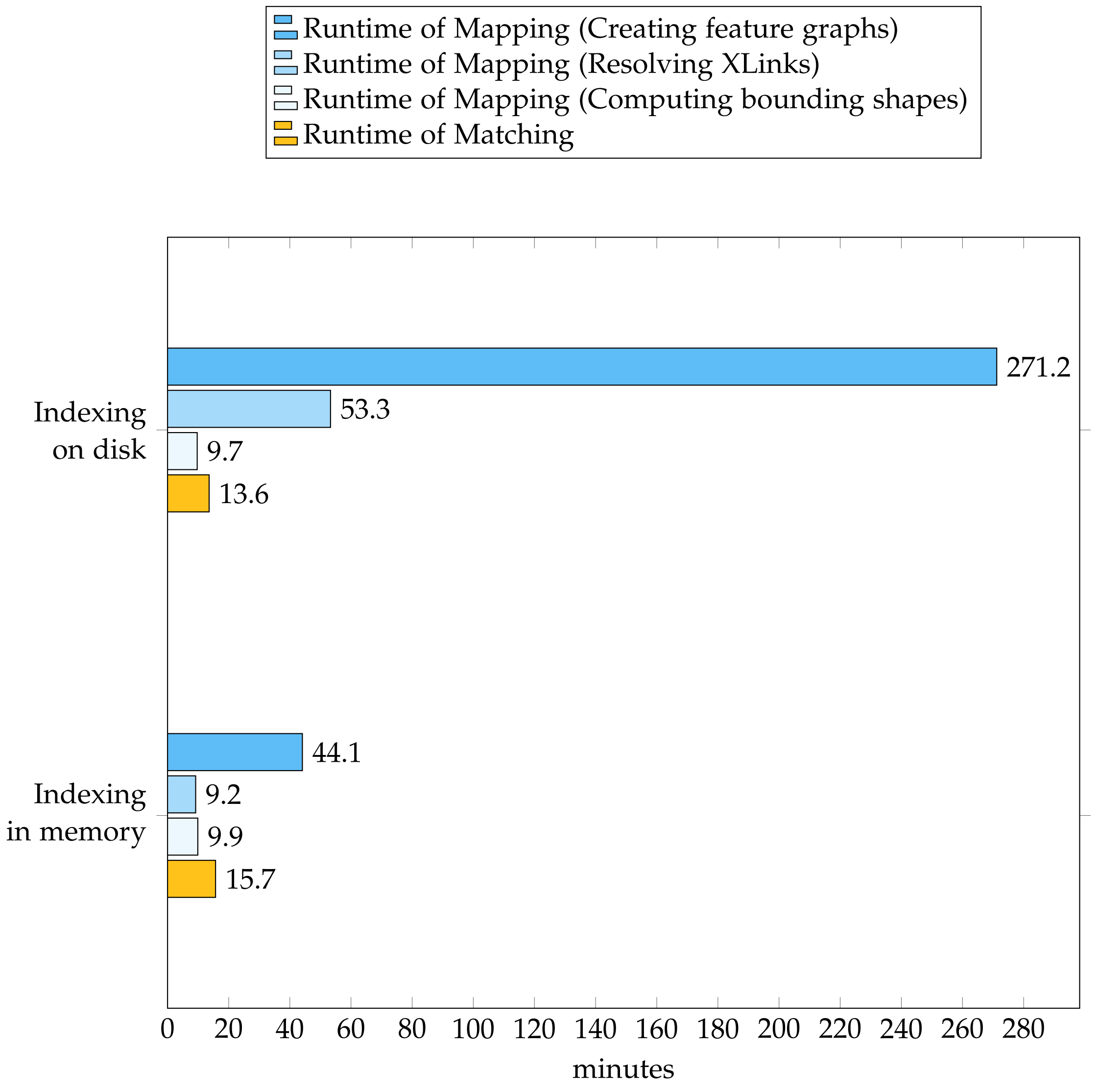


Figure 7.2: The impact of (built-in) indices in Neo4j on performance compared to an internal hash map used for indexing in main memory.

7.2.4 Differences in Performance between Building Batch Sizes

The effects on performance of applying different numbers of buildings wrapped in a batch transaction during the mapping and matching process are illustrated in Figure 7.3.

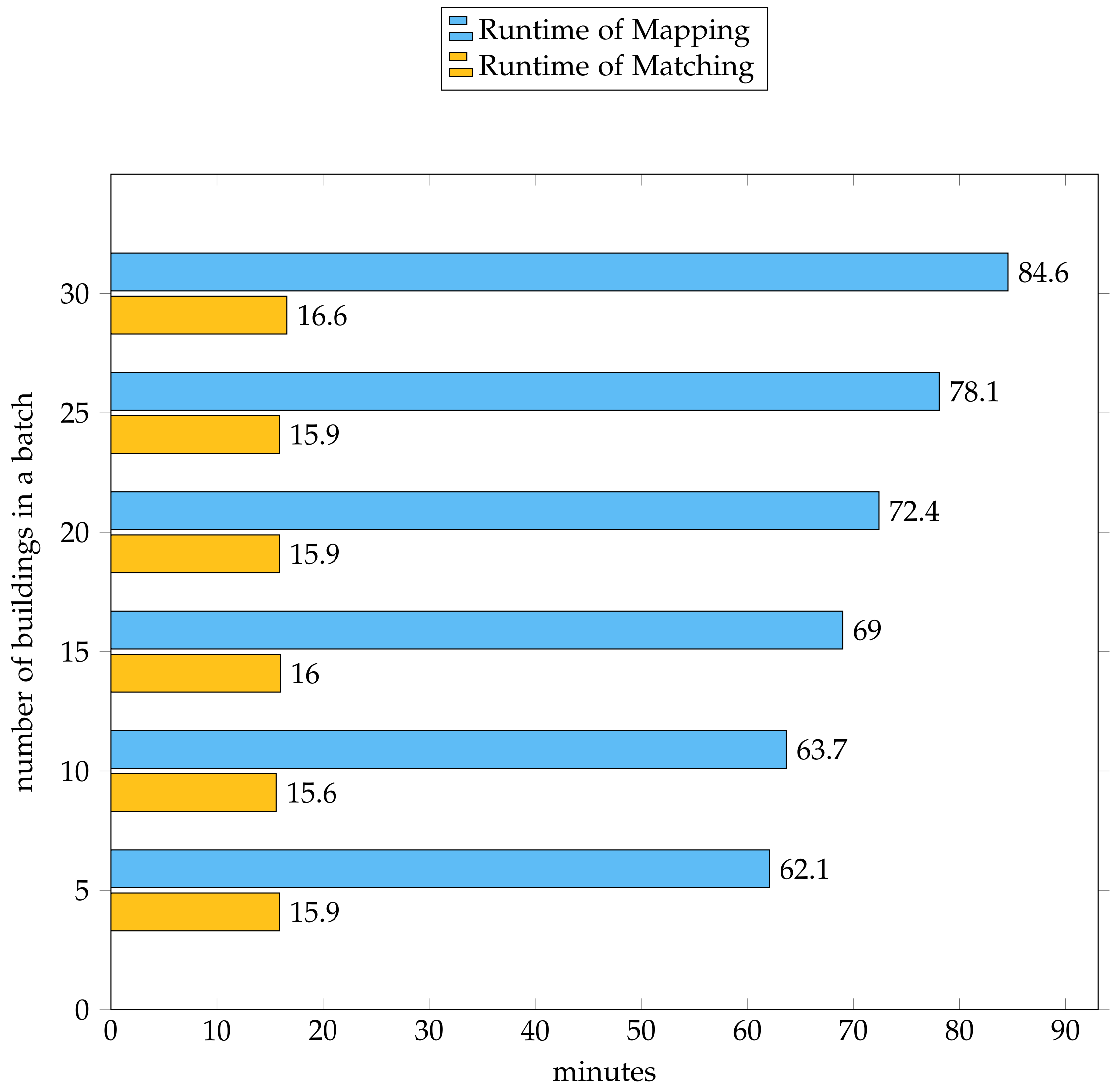


Figure 7.3: The effects on performance of applying different numbers of buildings wrapped in a batch transaction during the mapping and matching process.

7.2.5 Performance of the Grid Layout and R-tree

The performance differences in mapping and matching using the grid layout and R-tree approach are shown in Figure 7.4. Finally, Figures 7.5 and 7.6 visualize two R-tree images of the entire area of Berlin with values of *M* equal to 10 and 100 respectively.

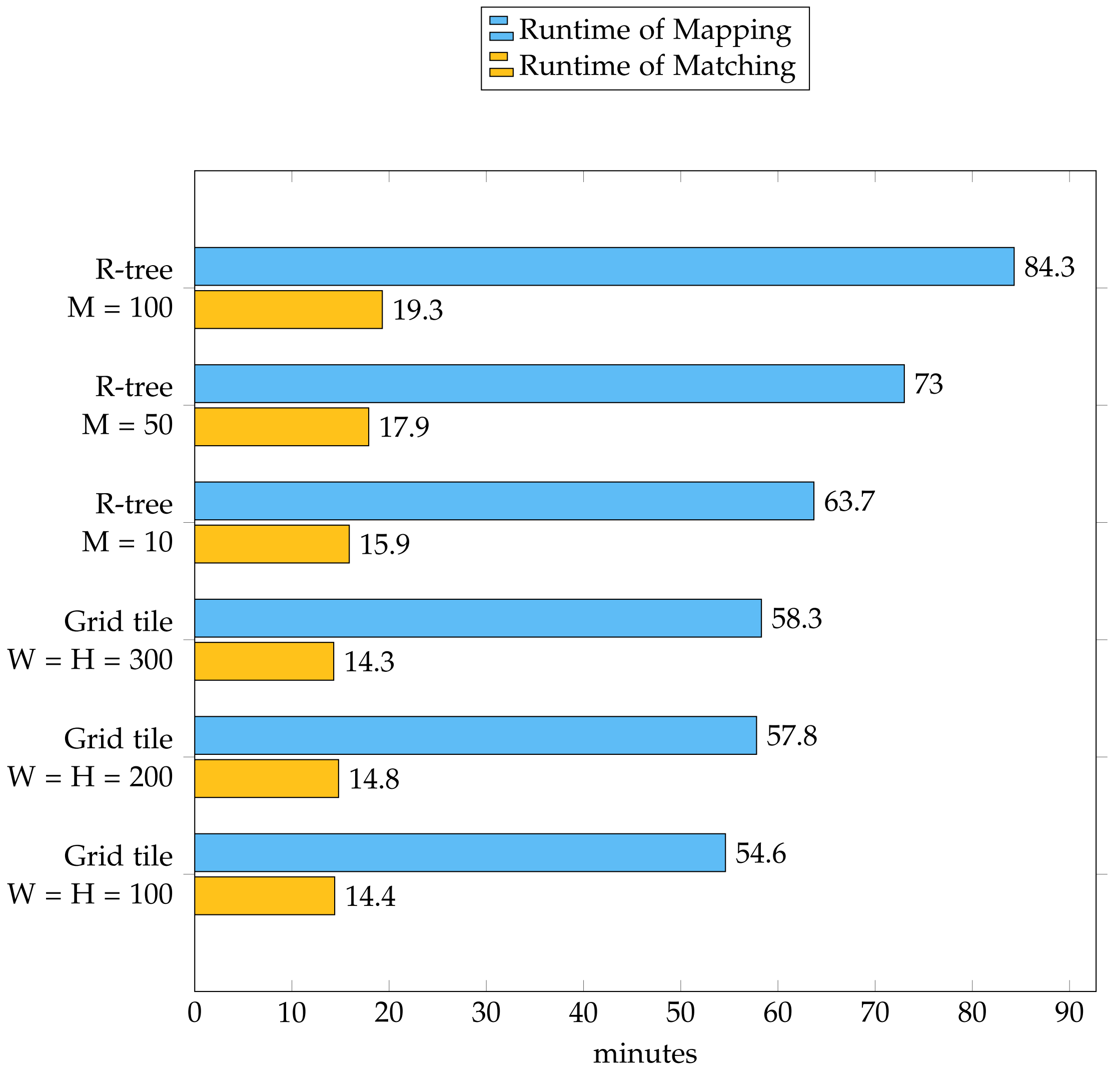


Figure 7.4: The performance differences in mapping and matching using the grid layout and R-tree approach, where *W*, *H* denote width and height of grid tiles respectively and *M* denotes the maximum number of entries per internal R-tree node.

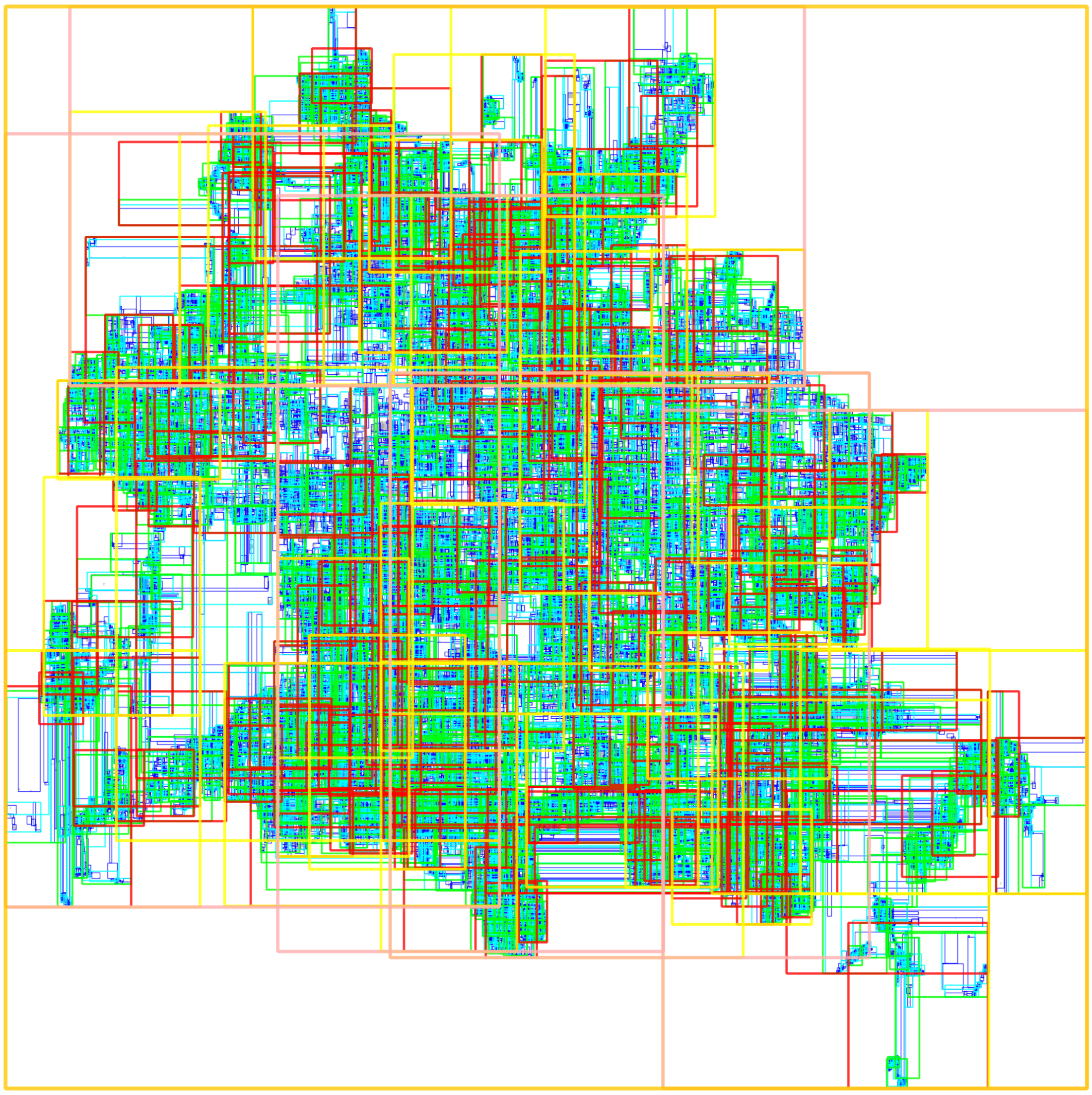


Figure 7.5: R-tree visualization with $M = 10$ of the entire area of Berlin. Each rectangle represents an R-tree node. Nodes of the same colour have the same height. Thus, this R-tree can be visualized as a tall but narrow-shaped tree.

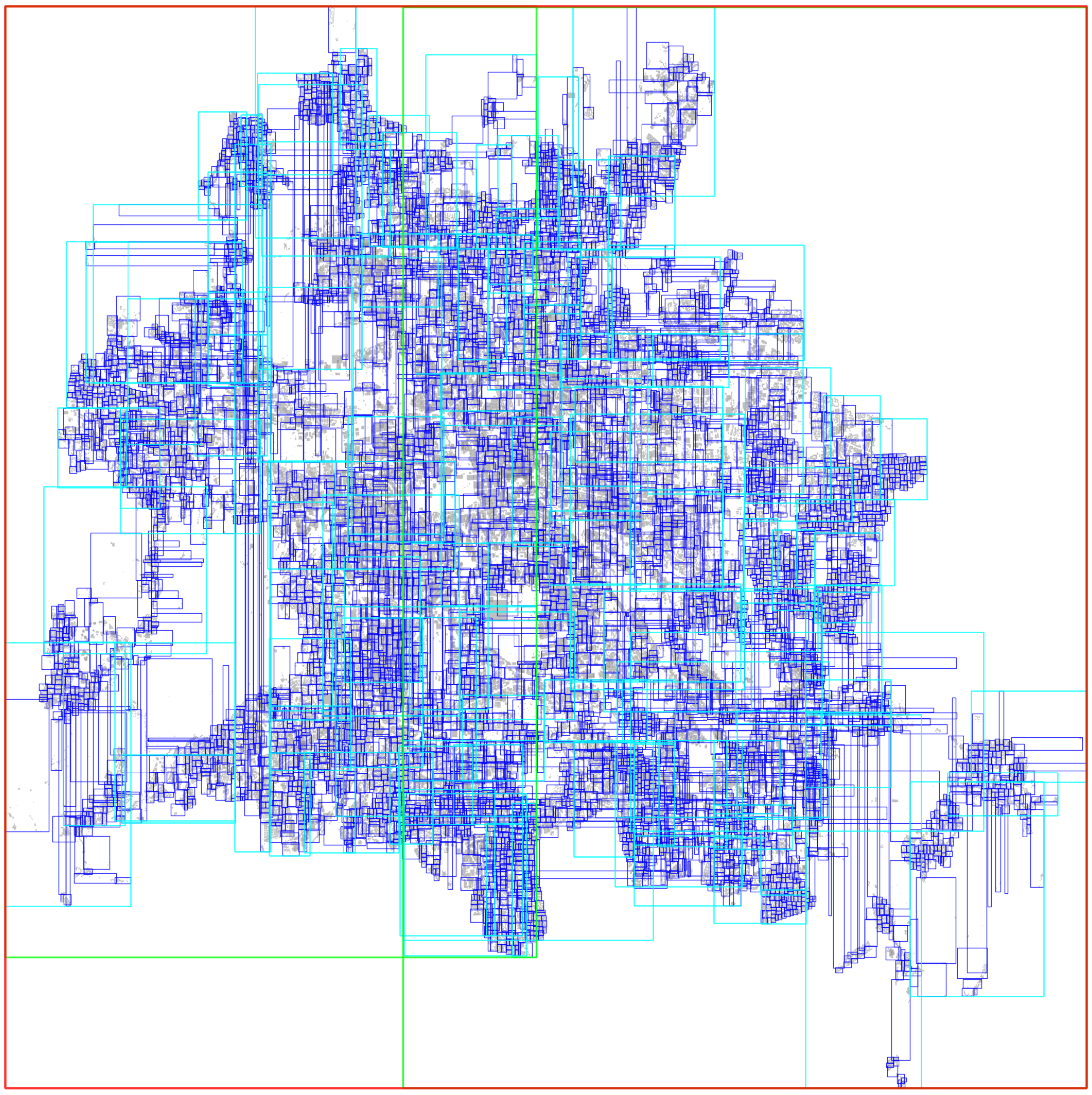


Figure 7.6: R-tree visualization with $M = 100$ of the entire area of Berlin. Each rectangle represents an R-tree node. Nodes of the same colour have the same height. Thus, this R-tree can be visualized as a shallow but broad-branched tree.

7.3 Discussion

The overall application performance depends on a number of factors tested in this chapter, namely:

Multi-threading In multi-core systems, executing the mapping and matching process concurrently often results in a great boost in performance. Figure 7.1 shows significant increases in both mapping and matching where multi-threading is applied. In addition, the matching process tends to benefit more from the number of used threads than mapping. For instance, an increase of 76% and 945% are observed in the mapping and matching process respectively going from single to multi-threaded with 1 producer and 15 consumers. However, diminishing returns are encountered if too many concurrent consumers (relatively to available physical CPU cores) are employed. For example, a negative performance of 71% of the mapping process is recorded in the work-pool of 63 consumers compared to 15 consumers, while the matching process remains almost constant. Thus, empirically, 1 producer and 15 consumers are recommended for systems with similar hardware. One producer for reading and writing data from and to disk is often sufficient, as these tasks do not benefit from concurrency.

Means of indexing Figure 7.2 shows that by using a self-developed indexing method using internal hash maps stored in main memory, the performance of the mapping process is accelerated to more than 5 times as fast as that of storing indices on disk. Therefore, indexing using hash maps is recommended in systems with sufficient main memory (e.g. an initial heap space of 30,000 MB of main memory is employed in this test). However, only the first two steps of the mapping process (i.e. creating feature graphs and resolving XLinks) are affected, others remain almost unchanged.

Building batch size Figure 7.3 shows that the greater (≥ 10) the number of buildings wrapped in a batch transaction is, the slower the mapping process becomes. On the other hand, the matching step remains virtually constant among various batch sizes. Thus, it is suggested to execute no more than 10 buildings per batch.

Spatial search strategies While the grid layout offers almost constant matching and slightly increasing mapping time as tile size grows, the performance of both processes using an R-tree depends noticeably on the maximum number of entries M per internal node as shown in Figure 7.4. Moreover, although the grid layout is generally faster in this experiment, it is advised to investigate buildings' spatial distribution beforehand. On the other hand, an R-tree can be applied for almost all scenarios while still giving decent performance for $M \leq 10$.

8 Conclusion and Future Work

Overall, the concepts and their implementations developed in this thesis have achieved the intended results mentioned in Chapter 1, namely:

- The mapping process is capable of handling arbitrarily large-sized CityGML documents given a reasonable amount of main memory and storage allocation. It facilitates the seamless interaction between unmarshalling CityGML elements to Java objects with the help of the library citygml4j and mapping Java objects to graph entities using the Neo4j Java Core API. The generated graph database represents its input CityGML instance documents with minimum data loss during the transformation of object-oriented Java objects to value-based graph entities in Neo4j.

- The matching process is generally capable of detecting arbitrary changes made manually to the old dataset. It can disambiguate most common syntactic ambiguities existing in GML, e.g. between XLink and in-line object declarations. Moreover, geometric objects such as points, line segments, polygons, surfaces, etc. are compared based on their geometrical properties given a predefined error tolerance. These entities can be matched correctly even with altered identifiers. Furthermore, buildings can be organized in a grid layout or an R-tree based on their spatial allocation. These strategies offer a noticeable boost in overall performance.

- Found deviations are attached to their respective sources in the graph database and can be transformed to WFS requests complying with the official OGC standards. In case of complex XML properties, such as CityGML generic attributes and external references, although the update procedures can be formally represented by graphs, the ordinary WFS is not expressive enough in such scenarios. Therefore, vendor-specific extensions allowed by the WFS standard, such as defined by the virtualcityWFS, can be employed.

While the current implementation is capable of mapping, comparing and updating changes reliably between arbitrary CityGML documents, improvements and extensions are possible in the near future. For instance, momentarily, only the module `Building` (with `Appearances`) is implemented. Other CityGML modules, like `CityFurniture`, `Transportation`, `Bridge`, `Tunnel`, etc. can be included in the future.

Moreover, it is previously assumed that both CityGML input documents are provided in the same spatial reference system, which is not always the case in practice. Therefore, one future task is to integrate the transformation between different spatial reference systems in the implementation.

Furthermore, the current assignments of buildings to grid tiles or R-tree nodes do not consider objects defined by implicit geometry. The calculation of buildings' bounding shapes can also be further optimized. A (reverse) parser capable of converting graph entities back to Java objects and CityGML elements correspondingly is of interest.

In addition, due to the technical limitations persisting in the current WFS versions, some sub-objects of building features cannot be updated using WFS transactions. However, this is subject to change as more extended specifications are released.

Finally, the methods and algorithms proposed in this research can be extended and applied to enable a version control system for collaborative work in modelling and storing digital 3D city models in the future as proposed by [Cha+15].

List of Figures

1.1 An overview of three major steps mapping, matching and updating of 3D city models using a graph database. 2

2.1 Official logo of CityGML. 7
2.2 Five different Levels of Details (LOD) 0 - 4 of a building representation in CityGML. 8
2.3 UML package diagram illustrating the separate modules of CityGML and their schema dependencies. 8
2.4 UML diagram of CityGML's building model. 9
2.5 An example of a web document and its DOM representation. 11
2.6 The JAXB binding process. 12
2.7 An example of the relational model. 14
2.8 Relevant connected data items from Figure 2.7. 15
2.9 An illustration of reading large CityGML datasets piece by piece. 17
2.10 An example of a graph representation of the data model shown in Figure 2.8 in Neo4j. 19
2.11 An example of the transaction management in Neo4j. 22
2.12 An example of an R-tree. 24
2.13 An R-tree representation of an excerpt from the CityGML dataset provided by the North Rhine-Westphalia state, Germany. 25
2.14 Illustration of an adapter connecting spatial indices in Neo4j Spatial with data already stored in Neo4j. 27
2.15 An overview of the Web Feature Service (WFS) in interaction with GIS clients and geospatial database. 28

3.1 An illustration of Steps 1 and 2 of the mapping process. 35
3.2 An illustration of Steps 3 and 4 of the mapping process. 36
3.3 UML class diagram of `RectifiedGrid` and its superclasses. 39
3.4 Result of mapping a `Grid` instance to a graph in the instance-based approach. 42
3.5 Result of mapping a `Grid` instance to a graph in the hierarchy-based approach. 46
3.6 An illustration of the splitting mechanism per feature in citygml4j. . . . 51

4.1 An illustration of the neighbourhood $N(\epsilon)$ of point P_1 in 2D and 3D. . . 60
4.2 An example of two geometrically matched `LineStrings` each consisting of three line segments. 61
4.3 Some examples illustrating the three requirements of `LinearRings`. . . . 63
4.4 An example of two perpendicular (planar) rings R_1, R_2 given normal vectors $\overrightarrow{n_1}$, $\overrightarrow{n_2}$ in three-dimensional space $Oxyz$. 64
4.5 Some examples illustrating the three requirements needed for interior boundaries of `Polygons`. 69
4.6 An example of two buildings, whose minimum bounding boxes have the same footprint but different heights. 71
4.7 An example of the shared volume of two minimum bounding boxes. . . 73
4.8 Examples of non-coincident and coincident grid layouts. 75
4.9 An illustration of grid construction. 76
4.10 An illustration of possible combinations that may occur while assigning buildings to tiles in a grid using its centre. 78
4.11 An illustration of matching city models by their grid layouts. 81

5.1 A UML class diagram of all edit operations. 86
5.2 An illustration of two geometrically equivalent polygons defined in two CityGML documents. 87
5.3 Two approaches to retrieving XML contents of a CityGML object. 94

6.1 An example of two producers and four consumers in the Producer-consumer design pattern. 108
6.2 An illustration of a deadlock encountered while concurrently matching buildings using a grid layout. 110

7.1 The differences in performance between the single-threaded and some combinations of different numbers of producers and consumers in the multi-threaded processing. 115
7.2 The impact of (built-in) indices in Neo4j on performance compared to an internal hash map used for indexing in main memory. 116
7.3 The effects on performance of applying different numbers of buildings wrapped in a batch transaction in the mapping and matching process. . 117
7.4 The performance differences in mapping and matching using the grid layout and R-tree approach. 118
7.5 R-tree visualization with $M = 10$ of the entire area of Berlin. 119
7.6 R-tree visualization with $M = 100$ of the entire area of Berlin. 120

List of Tables

2.1 An overview of WFS operations including their operation groups (indicated by colours), descriptions and request encodings. 32
2.2 WFS versions and their implemented operations. 33

3.1 Visible attributes and references of a `Grid` instance. 41
3.2 Visible attributes and references of a `RectifiedGrid` instance. 43

4.1 A list of point classes in citygml4j and their node labels in Neo4j. 59

7.1 Labelled nodes and their respective frequencies after two CityGML instances of the whole city Berlin are mapped. 114

List of Algorithms

1 `instance_based_map(instance)` 40
2 `hierarchy_based_map(instance)` 44
3 `hybrid_map(instance, container)` 48

4 `match_node(node1, node2)` 54
5 `match_properties(node1, node2)` 55
6 `match_relationships(node1, node2)` 57
7 `fuzzy_contains(other_area)` 67
8 `assign(building, grid)` 77
9 `match_grid(grid1, grid2)` 80
10 `match_rtree(city1, city2)` 83

11 `update_controller(city1, city2)` 92
12 `update(node)` 92

Listings

2.1 A simple Cypher query to retrieve building(s) with given ID. 20
2.2 An example of Neo4j Bolt in Java. 20
2.3 An example of Neo4j Java Core API. 21
2.4 An example of an HTTP POST request. 29

5.1 First sample CityGML document. 87
5.2 Second sample CityGML document. 89
5.3 WFS insert transaction of buildings. 93
5.4 WFS delete transaction of buildings. 95
5.5 WFS transaction of thematic properties. 96
5.6 WFS insert and update transaction of geometric properties. 98
5.7 WFS delete transaction of geometric properties. 98
5.8 WFS insert transaction of generic atributes. 99
5.9 WFS insert transaction of generic atribute sets. 100
5.10 WFS insert transaction of external references. 101
5.11 WFS delete transaction of generic attributes. 102
5.12 WFS update transaction of generic attributes. 102

Bibliography

[3DC16] 3DCityDB. *3D City Database for CityGML*. Version 3.3.0. Chair of Geoinformatics, Technische Universität München (TUMGI), virtualcitySYSTEMS GmbH, and M.O.S.S. Computer Grafik System GmbH. 2016. URL: `http://www.3dcitydb.org/3dcitydb/fileadmin/downloaddata/3DCityDB_Documentation_v3.3.pdf` (visited on 03/01/2017).

[3DC17] 3DCityDB. *citygml4j - The Open Source Java API for CityGML*. Mar. 2017. URL: `http://www.3dcitydb.net/3dcitydb/citygml4j/` (visited on 03/01/2017).

[Bac10] R. Baca. *R-tree example*. 2010. URL: `https://upload.wikimedia.org/wikipedia/commons/6/6f/R-tree.svg` (visited on 03/01/2017).

[Ber+08] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736, 9783540779735.

[Bil16] R. Bill. *Grundlagen der Geo-Informationssysteme. 6. völlig neu bearbeitete und erweiterte Auflage*. 6th ed. Wichmann, 2016. Chap. 3. Raum und Zeit in GIS, pp. 157–198. ISBN: 978-3-87907-607-9.

[Bra+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0*. Fifth edition. World Wide Web Consortium (W3C). Nov. 2008.

[CAM02] G. Cobena, S. Abiteboul, and A. Marian. “Detecting changes in XML documents.” In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. 2002, pp. 41–52. DOI: `10.1109/ICDE.2002.994696`.

[CG97] S. S. Chawathe and H. Garcia-Molina. “Meaningful Change Detection in Structured Data.” In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’97. Tucson, Arizona, USA: ACM, 1997, pp. 26–37. ISBN: 0-89791-911-4. DOI: `10.1145/253260.253266`.

[Cha+15] K. Chaturvedi, C. S. Smyth, G. Gesquière, T. Kutzner, and T. H. Kolbe. “Managing versions and history within semantic 3D city models for the next generation of CityGML.” en. In: *Selected papers from the 3D GeoInfo 2015 Conference*. Ed. by A. A. Rahman. Lecture Notes in Geoinformation and Cartography. Kuala Lumpur, Malaysia: Springer, 2015.

[Cha+96] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. "Change Detection in Hierarchically Structured Information." In: *SIGMOD Rec.* 25.2 (June 1996), pp. 493–504. ISSN: 0163-5808. DOI: `10.1145/235968.233366`.

[Cod70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: `10.1145/362384.362685`.

[Cox+04] S. Cox, P. Daisey, R. Lake, C. Portele, and A. Whiteside. *OpenGIS Geography Markup Language (GML) Implementation Specification*. Specification OGC 03-105r1. Version 3.1.1. Open Geospatial Consortium (OGC), 2004.

[DBE17] DB-Engines. *DB-Engines Ranking*. 2017. URL: `http://db-engines.com/en/ranking/graph+dbms` (visited on 03/01/2017).

[DeR+10] S. DeRose, E. Maler, D. Orchard, and N. Walsh. *XML Linking Language (XLink) Version 1.1*. World Wide Web Consortium (W3C). May 2010.

[EF91] M. J. Egenhofer and R. D. Franzosa. "Point-set topological spatial relations." In: *International Journal of Geographical Information Systems* 5.2 (1991), pp. 161–174. DOI: `10.1080/02693799108927841`. eprint: `http://dx.doi.org/10.1080/02693799108927841`.

[EH91] M. J. Egenhofer and J. Herring. *Categorizing binary topological relations between regions, lines, and points in geographic databases*. Technical report. Department of Surveying Engineering, University of Maine, 1991.

[Goe+06] B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, and T. Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006. ISBN: 0321349601.

[Grö+12] G. Gröger, T. H. Kolbe, C. Nagel, and K.-H. Häfele. *OpenGIS(R) City Geography Markup Language (CityGML) Encoding Standard*. Version: 2.0.0. Open Geospatial Consortium (OGC). Apr. 2012.

[Grö10] G. Gröger. *Modeling Guide for 3D Objects - Part 1: Basics (Rules for Validating GML Geometries in CityGML)*. Version 0.6.0. SIG3D - Special Interest Group 3D. Dec. 15, 2010. URL: `http://en.wiki.quality.sig3d.org/index.php/Modeling_Guide_for_3D_Objects_-_Part_1:_Basics_(Rules_for_Validating_GML_Geometries_in_CityGML)` (visited on 03/01/2017).

[Gut84] A. Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching." In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. Boston, Massachusetts: ACM, 1984, pp. 47–57. ISBN: 0-89791-128-8. DOI: `10.1145/602259.602266`.

[HM76] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison*. Computing Science Technical Report. Bell Laboratories, June 1976.

[Kes+15] A. van Kesteren, A. Gregor, Ms2ger, A. Russell, and R. Berjon. *W3C DOM4*. World Wide Web Consortium (W3C). Nov. 19, 2015. URL: `https://www.w3.org/TR/domcore/` (visited on 03/01/2017).

[KGP05] T. H. Kolbe, G. Gröger, and L. Plümer. "CityGML: Interoperable Access to 3D City Models." In: *Geo-information for Disaster Management*. Ed. by P. van Oosterom, S. Zlatanova, and E. M. Fendel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 883–899. ISBN: 978-3-540-27468-1. DOI: `10.1007/3-540-27468-5_63`.

[Man+05] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Springer Publishing Company, Incorporated, 2005. ISBN: 1852339772, 9781852339777.

[Nag17] C. Nagel. *citygml4j - The Open Source Java API for CityGML*. Version 2.4.3. 2017. URL: `https://github.com/citygml4j/citygml4j/` (visited on 03/01/2017).

[NBU10] G. Navratil, R. Bulbul, and A. U. Frank. "Maintainable 3D Models of Cities." In: *Proceedings of the 15 International Conference on Urban Planning, Regional Development and Information Society*. Real CORP, 2010, pp. 411–418.

[Neo17a] Neo Technology. *From Relational to Neo4j*. Neo Technology. 2017. URL: `https://neo4j.com/developer/graph-db-vs-rdbms/` (visited on 03/01/2017).

[Neo17b] Neo Technology. *The Neo4j Developer Manual*. Version 3.1. Neo Technology. 2017. URL: `http://neo4j.com/docs/developer-manual/current/` (visited on 03/01/2017).

[Neo17c] Neo Technology. *The Neo4j Java Developer Reference*. Version 3.1. Neo Technology. 2017. URL: `http://neo4j.com/docs/java-reference/current/` (visited on 03/01/2017).

[Neo17d] Neo Technology. *The Neo4j Operations Manual*. Version 3.1. Neo Technology. 2017. URL: `https://neo4j.com/docs/operations-manual/current/` (visited on 03/01/2017).

[Neo17e] Neo4j Spatial. *Neo4j Spatial v0.24-neo4j-3.1.1*. Version 0.24. Neo Technology. 2017. URL: `http://neo4j-contrib.github.io/spatial/` (visited on 03/01/2017).

[Ora15] Oracle Corporation. *Java Architecture for XML Binding (JAXB)*. Oracle Corporation. 2015. URL: `http://docs.oracle.com/javase/tutorial/jaxb/intro/arch.html` (visited on 03/01/2017).

[Ora16] Oracle Corporation. *Java HotSpot™Virtual Machine Performance Enhancements*. Version 8. Oracle Corporation. 2016. URL: `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html` (visited on 03/01/2017).

[Ora17] Oracle Corporation. *JAXB User's Guide*. Oracle Corporation. 2017. URL: `https://jaxb.java.net/2.2.11/docs/ch03.html` (visited on 03/01/2017).

[Red14] R. Redweik. "Semantic Change Detection for CityGML Documents." Master's thesis. Institute for Geodesy and Geoinformation Science, Technical University of Berlin, Oct. 2014.

[RPB09] S. Rönnau, G. Philipp, and U. M. Borghoff. "Efficient Change Control of XML Documents." In: *Proceedings of the 9th ACM Symposium on Document Engineering*. DocEng '09. Munich, Germany: ACM, 2009, pp. 3–12. ISBN: 978-1-60558-575-8. DOI: `10.1145/1600193.1600197`.

[Sel77] S. Selkow. "The tree-to-tree editing problem." In: *Information Processing Letters* 6.6 (Dec. 1977), pp. 184–186. ISSN: 00200190. DOI: `10.1016/0020-0190(77)90064-3`.

[SN09] M. Scholz and S. Niedermeier. *Java und XML: Grundlagen, Einsatz, Referenz*. Galileo Computing. Bonn, 2009. ISBN: 9783836213080.

[Tai79] K.-C. Tai. "The Tree-to-Tree Correction Problem." In: *J. ACM* 26.3 (July 1979), pp. 422–433. ISSN: 0004-5411. DOI: `10.1145/322139.322143`.

[vir16] virtualcitySYSTEMS. *virtualcityWFS 3.1.0 Manual*. Manual. Version 3.1.0. virtualcitySYSTEMS, Aug. 2016.

[Vre14] P. A. Vretanos. *OGC® Web Feature Service 2.0 Interface Standard*. OGC® Standard 09-025r2. Version 2.0.2. Open Geospatial Consortium (OGC), July 2014.

[WDC03] Y. Wang, D. J. DeWitt, and J. Y. Cai. "X-Diff: an effective change detection algorithm for XML documents." In: *Data Engineering, 2003. Proceedings. 19th International Conference on*. Mar. 2003, pp. 519–530. DOI: `10.1109/ICDE.2003.1260818`.

[ZS89] K. Zhang and D. Shasha. "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems." In: *SIAM J. Comput.* 18.6 (Dec. 1989), pp. 1245–1262. ISSN: 0097-5397. DOI: `10.1137/0218082`.