Technical University of Munich

SPATIO-SEMANTIC COMPARISON OF LARGE 3D CITY MODELS IN CITYGML USING A GRAPH DATABASE

Son H. Nguyen, Zhihang Yao, Thomas H. Kolbe

Abstract: The OGC open data model for the storage and exchange of virtual 3D city models City Geography Markup Language (CityGML) allows various syntactic ways to define a 3D city object. This on the one hand offers a high degree of flexibility in terms of creating new content-rich city models, but on the other hand complicates the automatic maintenance process of existing large CityGML documents. One often-stated example of such complications is the difficulty observed while attempting to detect possible thematic, geometrical as well as semantic deviations between two CityGML datasets of the same city. Existing studies have indicated that such problems can be solved using graph representations of CityGML documents. However, the question as how this concept can be realized still remains. Thus, this research provides an in-depth solution to this question in three main steps: (1) mapping two arbitrarily large-sized CityGML datasets efficiently onto graphs using a graph database (such as Neo4j), (2) matching mapped graphs based on concrete algorithms and attaching various types of EditOperations designed for updating the older CityGML dataset, and (3) executing attached EditOperations by converting them to transactions conforming to the Web Feature Service (WFS), the standard interface for updating geographical features across the web. The functionality and performance of the developed software is examined and demonstrated using the entire 3D city model of Berlin.

Keywords: 3D City Models, CityGML, spatio-semantic comparison, change detection, graph database, Neo4j, Web Feature Service

RÄUMLICH-SEMANTISCHER VERGLEICH GROSSER 3D-STADTMODELLE IN CITYGML UNTER VERWENDUNG EINER GRAPH-DATENBANK

Zusammenfassung: Der OGC-Standard zur Speicherung und zum Austausch virtueller 3D-Stadt- und Landschaftsmodelle City Geography Markup Language (CityGML) erlaubt zahlreiche syntaktische Varianten, wie 3D-Stadtobjekte in CityGML-Dokumenten repräsentiert werden können. Das bietet eine hohe Flexibilität beim Erzeugen von Stadtmodellen mit reichhaltigem Informationsgehalt, erschwert jedoch den automatisierten Wartungsprozess existierender großer CityGML-Dokumente. Ein prominentes, öfter vorkommendes Beispiel sind Schwierigkeiten beim Erkennen möglicher thematischer, geometrischer sowie semantischer Änderungen zwischen zwei CityGML-Datensätzen einer Stadt. Erste Arbeiten schlagen dazu vor, CityGML-Dokumente während ihres Vergleichs als Graphen darzustellen, aber es bleibt zumeist offen, wie dieses Konzept realisiert und effizient implementiert werden kann. Die hier vorgestellte Forschungsarbeit beantwortet diese Frage ausführlich in drei Hauptschritten: (1) das Abbilden zweier beliebig großer CityGML-Datensätze auf Graphen unter Verwendung einer Graphdatenbank (z. B. Neo4j), (2) das Vergleichen der abgebildeten Graphen und Erzeugen der verschiedenen Edit-Operationen, welche zum Aktualisieren des alten CityGML-Datensatzes benötigt werden, und (3) das Ausführen der eingefügten Edit-Operationen, indem sie zu Transaktionen der zur Aktualisierung der geographischen Features-Standardschnittstelle Web Feature Service (WFS) umgewandelt werden. Die Funktion und Performanz der entwickelten Software wird am Beispiel des kompletten 3D-Stadtmodells von Berlin untersucht und gezeigt.

Schlüsselwörter: 3D-Stadtmodelle, CityGML, räumlich-semantischer Vergleich, Veränderungsdetektion, Graphdatenbank, Neo4j, Web Feature Service

Autor(en)

M. Sc. Son H. Nguyen
M. Sc. Zhihang Yao
Univ.-Prof. Dr. rer. nat. Thomas H. Kolbe

Technical University of Munich
Chair of Geoinformatics
Arcisstraße 21
D-80333 München

E: son.nguyen@tum.de
zhihang.yao@tum.de
thomas.kolbe@tum.de

1 INTRODUCTION

As an official OGC standard for the storage and exchange of virtual 3D city models, CityGML is capable of describing most common 3D city objects (such as buildings, bridges, tunnels, vegetation, traffic, etc.) and has been employed in a wide range of different areas, from urban planning and facility management to environmental simulations and thematic inquiries. One of the main factors contributing to this success is that, unlike conventional modelling tools that can only store the 3D geometry and graphical appearances of 3D urban objects for pure visualization purposes, CityGML also describes them in five different levels of details (LODs 0-4) and includes their semantic as well as thematic properties (such as relationships between objects and object attributes respectively), since "one of the most important design principles for CityGML is the coherent modelling of semantics and geometrical/topological properties" (Gröger et al. 2012). Moreover, CityGML allows multiple geometrical and syntactic ways to define a 3D city object offering a high degree of flexibility. For instance, two geometrically equivalent wall surfaces, on the one hand, can be represented by a single polygon or a set of smaller polygons, and on the other hand, can be defined in different syntactic ways, e.g. as in-line objects, or as references referring to other existing walls of adjacent buildings via the XML Linking Language (XLink). This flexibility is "especially important with respect to the cost-effective sustainable maintenance of 3D city models" (Gröger et al. 2012), as it ensures CityGML documents can be shared over various applications that make use of the model's common semantic information.

However, the facts that (1) geometrical and syntactic ambiguities may exist in CityGML datasets, (2) CityGML elements belong to a complex hierarchical structure, and (3) CityGML documents can become very large in size have proved to be major challenges to maintain sustainable 3D city models. One often-stated example is the difficulty observed while handling undocumented collaborative and chronological changes of an existing city model (see Figure 1). Such changes are inevitable, since as cities evolve over time, so does the need to adjust their models accordingly (Navratil et al. 2010). Furthermore, because the current state of CityGML does not support version control for tracking changes, multiple model documents of the same city may accumulate over time. As a result, during the maintenance phase, old city datasets are often overwritten completely with newer ones, which not only causes a large number of unnecessary transactions, but also loses extended thematic data that was assigned to the older version of the 3D model during the given time period.

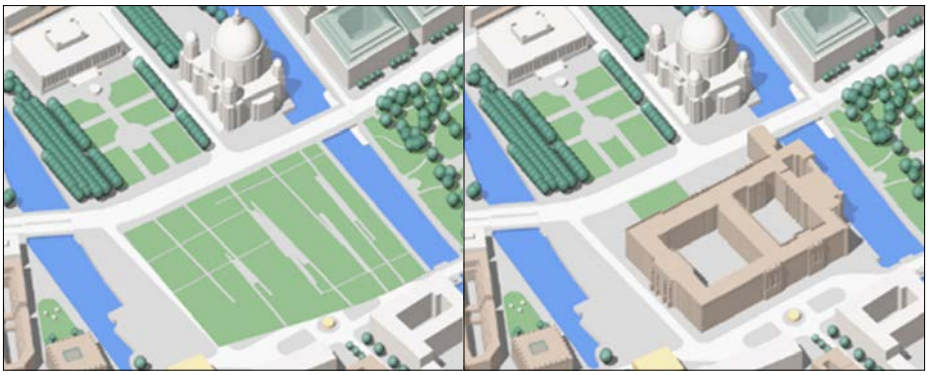


Figure 1: An example of chronological changes (e.g. a new building has been constructed) between two models of the same city recorded at different timestamps (Source: The City of Berlin)

Therefore, instead of replacing older records, an ideal solution should first compare the models, and then attach edit operations on the fly to detected deviation sources. Such edit operations represent real changes between datasets and can be utilized to commit transactions in the database. This way, older datasets can both be updated and still retain their respective core structure, including their syntactic structure and internal object references. This plays a key role in enabling a version control system for collaborative work in modelling and storing digital 3D city models (Chaturvedi et al. 2015). Moreover, the number of transactions required for e.g. a WFS-enabled database is also reduced significantly, since only real changes are committed.

In order to achieve this, considering the facts that CityGML elements belong to a graph-like structure and thus both geometrical and semantic ambiguities can theoretically be disambiguated using a graph, this research addresses the above-mentioned major challenges and proposes an approach to detect spatio-semantic changes in arbitrarily large-sized CityGML datasets utilizing a graph database. This approach can be applied to almost all available popular graph databases. The graph database Neo4j is employed in this research, as it provides relatively comprehensive API documentation. In addition, its community version is open-source and can be employed free of charge. Basic support for spatial data representation and indexing is also available using the plug-in Neo4j Spatial.

Please note that this article is a substantially extended version of Nguyen et al. (2017).

2 RELATED WORK

Existing conventional *diff* tools, such as the Hunt-McIlroy algorithm (Hunt & McIlroy 1976), can only detect changes in pure texts and is therefore incapable of handling highly structured data models like CityGML. Bakillah et al. (2009) proposed a conceptual basis for a semantic similarity model (Sim-Net) for ad hoc network based on the multi-view paradigm. Olteanu et al. (2006) addressed the automatic matching of imperfect geospatial data during database integration. However, since both of these researches mainly focused on either the semantic or geometrical aspect of city objects, they are not fully applicable to CityGML, which provides an integrated view of both aspects.

Later, Redweik & Becker (2015) presented a concept for detecting semantic and geometrical changes in CityGML documents. Since CityGML is an application schema of XML, which is a tree data structure, by assuming that CityGML instances can also be represented as trees, they extended the algorithm "X-Diff" (Wang et al. 2003) that considers tree equivalence as isomorphism. However, in

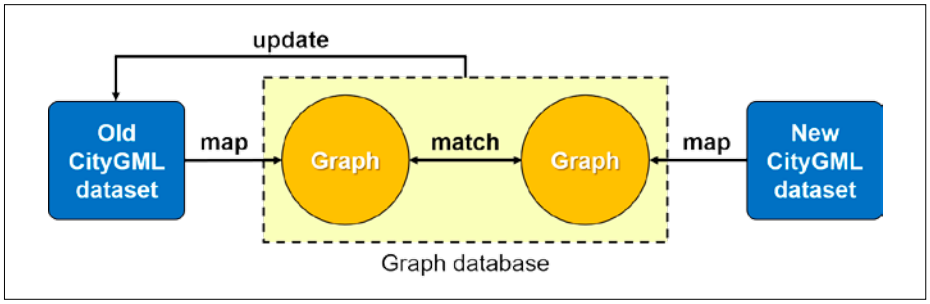


Figure 2: An overview of three major steps mapping, matching and updating of 3D city models using a graph database

contrast to XML, CityGML is not a tree but a graph data structure by definition, as it may contain cycles and nodes linked by multiple parents (e. g. due to XLinks). Therefore, this approach is generally not expressive enough considering CityGML's graph data structure. Moreover, the methods proposed by Redweik & Becker (2015) are not yet evaluated against massive input datasets.

Falkowski & Ebert (2009) introduced a graph-based schema for integrated models of urban data encoded in CityGML using the TGraph technology. Their approach shows how geometric, topological, semantic and appearance information can be stored, managed and processed in one integrated graph model and thus "forms the basis for the application of efficient graph-matching algorithms" in the context of object-recognition. Later, Agoub et al. (2016) addressed some of the biggest limitations of storing and managing object-oriented OGC data models (e. g. CityGML) inside a spatial database, such as most Relational Database Management Systems (RDBMS) are rather suitable to "flat data structure" and "mapping object-oriented data models into compact relational schemas without losing information is a challenging task". They then introduced a lightweight mapping approach that supports on-the-fly mapping and storage for various OGC standards (i. e. SensorML, CityGML, etc.) into the graph database Neo4j and ArangoDB. Both concepts provided by Falkowski & Ebert (2009) and Agoub et al. (2016) are promising as they show the potential of using graphs to represent highly complex hierarchical data structures. However, the introduced methods are rather a proof of concept as they do not cover in details how complex CityGML objects with their inheritance information and references (like XLinks) can be fully mapped and compared using graphs. Their implementations were also not designed to process massive input datasets.

To deal with large input datasets, it is of great advantage to efficiently preselect potential matching candidates based on their geometrical/topological properties. Objects' topologically relative allocations can be expressed by the "4" or "9-intersection model" ("4-IM" or "9-IM") (Egenhofer & Franzosa 1991, Egenhofer & Herring 1991). In addition, an object can be localized by recursively dividing its parent graph into quadtrees (2D) or octrees (3D) and colouring their interior as well as exterior (Berg et al. 2008). Alternatively, an R-tree can be applied to spatial objects grouped in regions based on their topological properties (Guttman 1984). Since R-trees are balanced, their query response time in logarithmic time complexity $O(\log_M n)$ is particularly efficient in large databases, where M is the maximum number of entries allowed per internal node and n is the number of nodes in the tree.

3 MAPPING 3D CITY MODELS ONTO A GRAPH DATABASE

Figure 2 shows the overall workflow, which is divided into three main steps: (1) mapping two arbitrarily large-sized CityGML datasets efficiently onto graphs, (2) matching mapped graphs based on concrete algorithms and attaching various types of *EditOperations* designed for updating the older CityGML dataset, and (3) transforming attached *EditOperations* to transactions conforming to the Web Feature Service (WFS). The graph database Neo4j is employed throughout the implementation. In

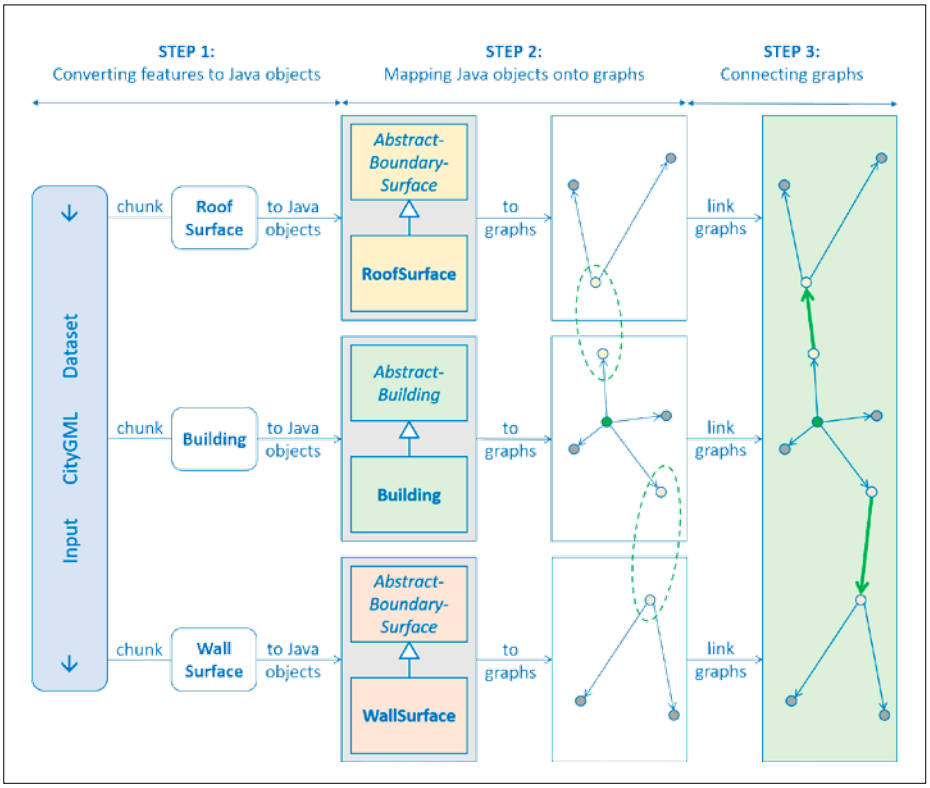


Figure 3: An overview of the mapping process. In this example, a building and its boundary surfaces, each contained in a chunk held in main memory, are converted to Java objects first using the library citygml4j (step 1). In step 2, these Java objects are mapped onto graphs correspondingly using the Neo4j Java Core API and self-developed algorithms. Finally, in step 3, mapped graphs are connected to each other using XLinks to form a unique and fully connected graph representation of the input building.

Neo4j, each city object is stored as a graph node, while the relationships between these objects are represented as edges between nodes. In other words, nodes are connected directly to each other. This is particularly useful in data models that have a complex and multi-level deep hierarchical structure like CityGML. In this chapter, the mapping of city objects onto Neo4j graphs shall be explained in several four smaller steps:

1. Reading CityGML datasets and converting features to Java objects;
2. Mapping Java objects onto graphs;
3. Connecting mapped graphs using XLinks;
4. Calculate minimum bounding boxes of buildings.

An overview of the first three steps can be found in Figure 3. Step 4 solely serves as a preparation for the matching process.

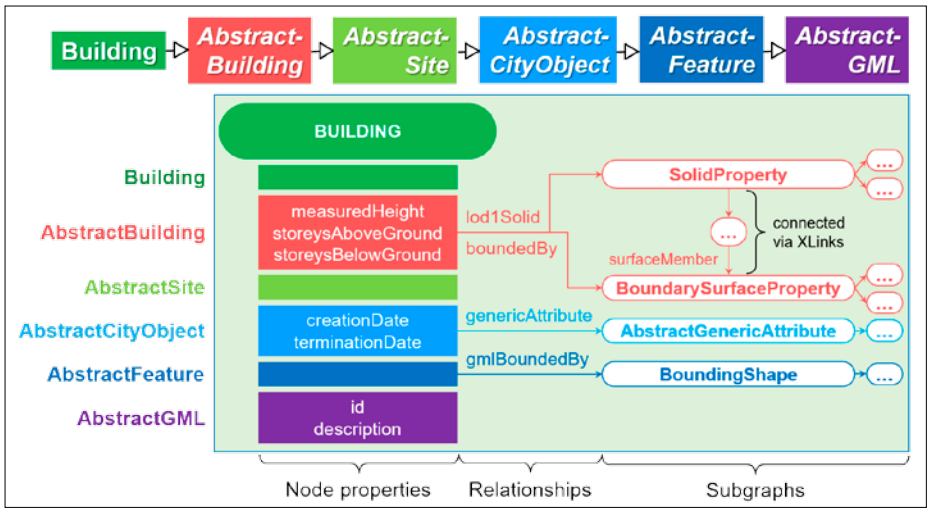


Figure 5: An example of a graph representing a Building object. Rounded rectangles represent nodes. Node properties are displayed below the node labels in rectangles. The colours indicate the originating classes, in which nodes and properties are defined. Relationships are shown as (directed) arrows. Subgraphs on the right-hand side are graph representations of complex Java object attributes of the current building. The container BUILDING node is expanded (i. e. gains new properties and subgraphs) successively for each superclass in the class hierarchy.

3.1 READING CITYGML DATASETS IN JAVA

CityGML documents can be processed with the help of various XML parsing APIs in Java such as the Document Object Model (DOM), Java Architecture for XML Binding (JAXB), Simple API for XML (SAX) or Streaming API for XML (StAX). Each API comes with their own advantages and disadvantages depending on the application domain. Considering the fact that CityGML datasets have highly complex hierarchical structure and can grow quickly in size, the library *citygml4j* is employed. It utilizes a combination of JAXB and SAX (Nagel 2017), which allows partial unmarshalling (or deserialization) of CityGML elements into Java objects with efficient memory consumption (see Figure 4). This is achieved by dividing the input datasets into smaller chunks (or pieces), each of which is a feature such as boundary surface or a top-level feature such as building (see Figure 3, Step 1). Moreover, this approach provides an object-oriented view of read CityGML data, which facilitates the transformation of unmarshalled Java objects to graph entities in the next step.

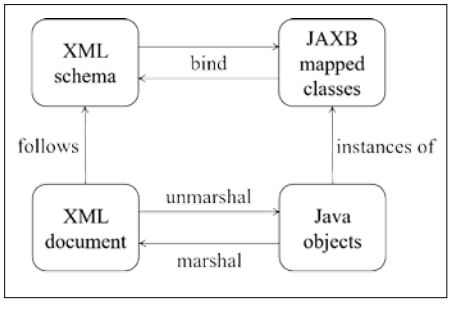


Figure 4: The JAXB binding process (Adapted from Oracle Corporation 2015)

3.2 CONVERTING JAVA OBJECTS TO GRAPH ENTITIES

Java objects unmarshalled by the library citygml4j in the previous step are now transformed to corresponding graph entities in Neo4j using the *Neo4j Java Core API* (see Figure 3, step 2). Conceptually however, two major challenges arise. Firstly, unmarshalled Java instances belong to a complex and multi-level deep class hierarchy defined by the XML schema of CityGML. This poses the difficulty in designing suitable graph structures that are capable of not only representing different instances of the same class efficiently, but also handling polymorphism correctly, where an instance of a superclass can be replaced by those of its subclasses. For instance, the aggregation *boundedBy* in the class *AbstractBuilding* requires an *AbstractBoundarySurface* object, which can be a *RoofSurface*, *WallSurface*, *GroundSurface*, etc. Secondly, Neo4j is a value-based graph database, which means that no explicit schema modelling (incl. inheritance relationships) is possible. As a result, mapping Java objects onto graphs without losing any information, particularly their hierarchical inheritance relations, is difficult.

To resolve these challenges, a new approach capable of creating graph representations of given Java objects using their hierarchical information is proposed (see Algorithm 1). The key concept is the use of a central expandable container node, where all (i. e. own and inherited) attributes and references of the respective Java object can be appended successively for each superclass. Namely, the mapping

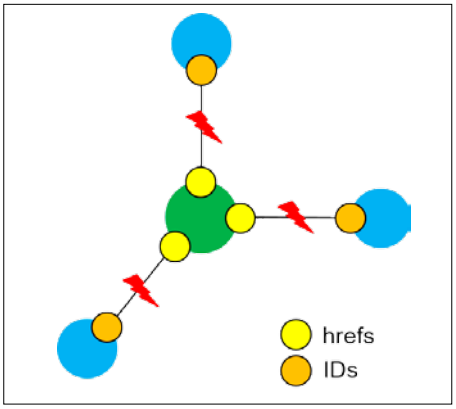


Figure 6: Explicit connections between features (blue circles) and their parent (green circle) are severed during the splitting process using citygml4j in step 1. To enable subsequent reconstruction of such lost connections, hrefs or XLinks (yellow circles) containing the IDs (orange circles) of split features are automatically generated and attached to the affected parent element.

process starts with an empty container node filled with the contents defined in the lowermost Java class in the class hierarchy. Then, this container is successively expanded with the contents of each superclass while moving up the class hierarchy.

The structure correspondence between Java objects and their graph representations assigned by Algorithm 1 are listed in Box 1.

An illustration of a graph representation of a building is shown in Figure 5.

The advantages of this approach are:

- Efficient and sustainable implementation;
- Compact but expressive resulting graphs;
- Minimum to zero information loss while mapping.

3.3 CONNECTING MAPPED CITY OBJECTS USING XLINKS

XLink is a simple yet practical means to referencing or reusing existing elements without having to define them "in-line" repeatedly and thus essentially reduces redundancies in XML documents (Bray et al. 2008, DeRose et al. 2010). However, despite their syntactic differences, both XLink and in-line declaration are often used to effectively define the same objects.

In the mapping process, XLinks can be found due to (1) user input, i.e. XLink references already exist in the input CityGML datasets before the mapping process is started (such as XLinks referring to existing building boundary surfaces listed in a Solid object); or (2) the splitting mechanism applied during the mapping process to divide large CityGML datasets into smaller chunks as described in step 1 (Section 3.1). In both cases however, these XLinks must be resolved to produce unique and fully connected graphs for the matching process. In the latter case, each time a feature (e.g. *WallSurface*) is split from a building (i.e. their connection is lost) while streaming the input CityGML document, an XLink object containing the ID of the split feature is automatically created and attached to the affected parent building by the tool citygml4j (see Figure 6 as well as the missing links marked by the green ellipses and arrows in Figure 3, step 2 and 3 respectively). This allows the subsequent recovery of such severed connections caused by the splitting process.

Box 1

City Object Structure in Java	**Corresponding Graph Structure**
(Complex) Instance or Object	Node
Instance type (e.g. of class *Building*)	Node label (e.g. *BUILDING*)
Inheritance (e.g. class *Building* inherits *AbstractBuilding*)	No explicit inheritance possible; instead, contents of all superclasses are successively added as node attributes (if they are simple texts) or attached as subgraphs (if they are complex) to the main container node
Simple (pure text) object attributes (e.g. building's year of construction)	Node attributes (e.g. *buildingNode.yearOfContruction*)
Complex object attributes or references (e.g. boundary surfaces of a building)	Two components: (1) A subgraph representing the object, and (2) a relationship edge that connects the main container node to this subgraph (e.g. the subgraph representation of a *WallSurface* is connected to the building node by a relationship called *boundedBy*)

Algorithm 1: `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 via map(attribute, null);
12         create a relationship from container to child;
13     end
14 end
15 if instance inherits SuperClass then
16     call map((SuperClass) instance, container);
17 end
18 return container;
```

Algorithm 1: Mapping a Java object onto corresponding graph entities

By employing a graph database, not only can such lost connections between features and their respective parents be reconstructed, but the syntactic ambiguities between in-line and XLink objects can also be disambiguated. This is realized in two different approaches using internal hash maps held in memory or Neo4j's built-in indices stored on disk. Each time a node containing an ID or href is encountered during the mapping process, a corresponding entry is stored in the respective index structure (in other words, all XLinks regardless of their usage shall be recorded). Then, after

all feature chunks have been mapped onto graphs, a "post-processing" searches for indexed hrefs and IDs and links them together accordingly. As a result, a unique and fully connected graph representation of a given city object is formed (see example in Figure 5, where the *BoundarySurfaceProperty* is connected to both its parents the *Building* and the *Solid* object due to XLinks). This graph is an unambiguous representation of a CityGML dataset independent from the many syntactic variations of respective CityGML files.

Internal hash maps offer fast response time but come at the cost of memory consumption. On the contrary, Neo4j indices require less memory but may slow down the mapping process due to costly disk read and write operations. Thus, the latter approach depends greatly on the storage selections (e.g. a Solid State Drive (SSD) is typically significantly faster in terms of read and write speed compared to a Hard Disk Drive (HDD)).

3.4 CALCULATING MINIMUM BOUNDING BOXES OF BUILDINGS

By default, the library citygml4j provides a built-in function that can compute the minimum bounding box of a spatial Java city object (e.g. a building) by considering all of its geometric contents (e.g. boundary surfaces). However, this method has some limitations. Firstly, input Java objects must be completely available in memory as a whole, which is not always the case, since Section 3.1 shows that large CityGML datasets are to be split into smaller chunks that are successively loaded into main memory. Secondly, if Java objects have unresolvable XLinks (such as those contained in not yet loaded feature chunks), the function may fail. Thus, to overcome these limitations, graph representations, which are now connected and syntactically disambiguated as a result of Section 3.3, are reversely transformed to Java objects, from which respective minimum bounding boxes can then be calculated using the above-mentioned built-in function.

4 MATCHING 3D CITY MODELS USING GRAPH DATABASE

The mapping process in the previous step produces unique and fully connected graph representations of two arbitrarily large-sized input CityGML models. This chapter explains how these graphs can be semantically and geometrically compared to each other. Since graphs are composed of nodes and relationships, the matching process is based around the concept of their structure. Namely, it matches the entire two given graphs from top to bottom (i.e. from root to leaf nodes) in the following order: node, node properties, relationships and corresponding subgraphs. Subgraphs are matched accordingly using the same method (recursion). Semantically, the matching process only allows the comparison of the following entities if they are:

- Nodes of the same type, i.e. having the same label;
- Node properties of the same name;
- Relationships of the same type;
- Subgraphs pointed from the same relationship type.

Subgraphs representing geometric objects (such as points, lines, polygons, etc.) are additionally matched based on their geometric types and properties, which shall be discussed in more details in the following sections.

Algorithm 2: `match_relationships(node1, node2)`

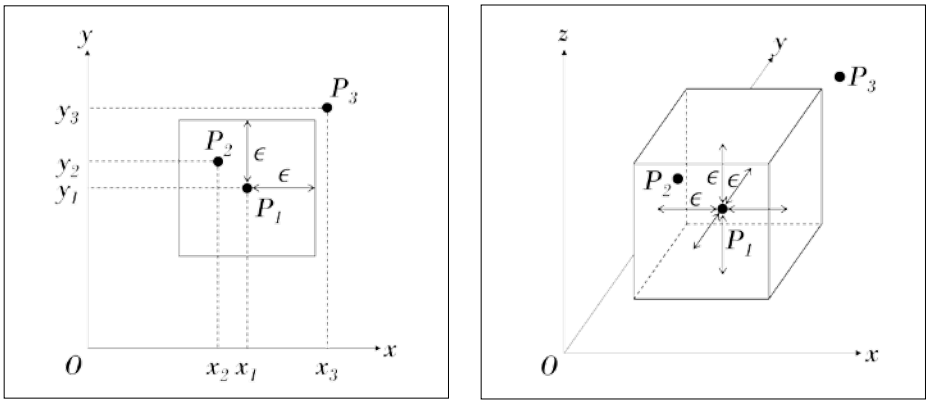
Input : *node1* and *node2* of graphs representing nodes of old and new city model respectively

```
1  foreach matched rel_type of node1 and node2 do
2      chdr1 ← node1.get_children(rel_type);
3      chdr2 ← node2.get_children(rel_type);
4      foreach child1 of chdr1 do
5          child2 ← chdr2.find_candidate(child1);
6          if child2 is not empty then
7              match_node(child1, child2);
8          end
9      end
10     foreach unmatched child1 of chdr1 do
11         create a DELETE operation;
12     end
13     foreach unmatched child2 of chdr2 do
14         create an INSERT operation;
15     end
16 end
17 foreach unprocessed rel_type of node1 do
18     chdr1 ← node1.get_children(rel_type);
19     foreach child1 of chdr1 do
20         create a DELETE operation;
21     end
22 end
23 foreach unprocessed rel_type of node2 do
24     chdr2 ← node2.get_children(rel_type);
25     foreach child2 of chdr2 do
26         create an INSERT operation;
27     end
28 end
```

Algorithm 2: Matching relationships of two given nodes in the graph database

4.1 COMPARING NODE PROPERTIES

Actual data are mostly stored in node properties. Thus, differences found in node properties indicate possible deviations of respective data sources. In Neo4j, node properties are identified by their unique name and thus values of equally named properties are to be compared with one another. Unmatched properties remaining after the process is complete indicate that they are either removed from the older model or inserted into the newer model. To model such changes (i.e. update, delete and insert on both property and node level) and enable subsequent transactions to update the older model, each deviation found is attached with an *EditOperation* graph node on the fly, which stores all relevant information such as name of affected proper-



a) $N(\epsilon)$ as a square in 2D

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

Figure 7: An illustration of the neighbourhood $N(\epsilon)$ of a reference point P_1 in 2D (a) and 3D (b). Since P_2 is located inside of $N(\epsilon)$, it is matched with P_1. On the other hand, P_3 is not matched with P_1 as it is located outside of $N(\epsilon)$.

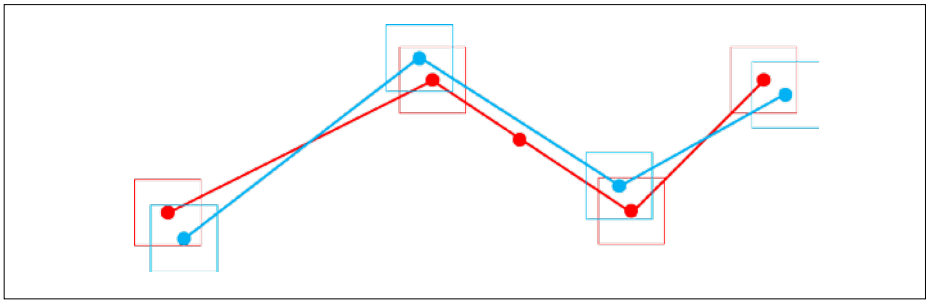


Figure 8: An example of two geometrically matched LineStrings with error tolerances taken into account, regardless of the fact that the red LineString has two consecutive collinear line segments

ties, old and new property values, etc. for later use. An overview of all edit operations is given in Section 6.1.

4.2 MATCHING NODE RELATIONSHIPS

Compared to node properties, matching relationships between two given nodes is more complex considering the fact that relationships in Neo4j can be traversed in both directions, namely in *OUTGOING* and *INCOMING* direction. The matching process must however remain consistent in one specific traversing direction, so that no node is processed twice. The chosen direction is *OUTGOING*, as the matching process starts with root nodes. Additionally, in contrast to node properties, a relationship may occur multiple times for a given node (i.e. 1 to n, n to 1 and n to m relationships).

Taking these into account, Algorithm 2 describes the main concept of matching relationships of two given nodes, where the function *find_candidate* in Line 5 plays a decisive role in terms of both efficiency and correctness of the whole matching process, as it determines which object pairs should be compared to one another based on their specific characteristics. In CityGML, the most important aspects that can be used as a matching pattern among objects are their geometrical properties as well as spatial extents.

4.2.1 MATCHING POINT GEOMETRIES

Points are a primitive notion, upon which all other geometric objects are built. Since points do not have length, area or volume, the only property employed to distinguish them from others is their coordinates (mostly in 2D or 3D). In practice, however, real-world coordinates of the same point location may differ if they are given in different Spatial Reference Systems (SRS). Therefore, input CityGML instance documents should first be provided in the same spatial reference system before they can be matched.

On the other hand, even provided in one common 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 error tolerance ϵ, it is often sufficient to compare individual point coordinates in each dimension, which requires only subtractions. In this case, $N(\epsilon)$ shall be a square (2D) or a cube (3D) (see Figure 7). 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. Two geometrically matched points are equal and thus no further comparison is needed.

4.2.2 MATCHING THE GEOMETRY OF LINE SEGMENTS

Since line segments (or *LineStrings*) are composed of points, they can be geometrically matched by iterating over all control points and examining their spatial similarities successively with error tolerance ϵ taken into account (see Figure 8). Consecutive collinear line segments (given an empirically predetermined distance tolerance) can be merged together and thus treated as a single segment during matching. Alternatively, two *LineStrings* can be matched using the Buffer Overlay Statistics (BOS) methods (Tveite 1999). Like points, geometrically matched *LineStrings* are also considered equal.

A more general concept of *LineStrings* is curves. A curve has a positive orientation and each of its curve segments may have a different interpolation method. However, as long as such curves are composed of points, the same approach can be applied assuming the respective interpolation methods are also identical, i.e. they must also be checked.

4.2.3 MATCHING THE GEOMETRY OF 3D RINGS

A ring in CityGML can be thought of as a closed *LineString* described in Section 4.2.2. Buildings in CityGML make extensive use of polygons (Section 4.2.4), whose boundaries are typically represented as *LinearRings* (Cox et al. 2004,

Gröger et al. 2012, Gröger 2010). Although a *LinearRing* can theoretically consist of nonplanar points in 3D, only *LinearRings* containing coplanar points are considered. The geometric comparison of rings can be performed with the aid of the libraries Abstract Window Toolkit (AWT) or Java Topology Suite (JTS). However, both of them are only applicable to geometric objects in 2D space, while rings are arbitrarily oriented in 3D. Therefore, only rings that have similar orientations (given an empirically predetermined angle tolerance between their normal vectors) and near-zero plane-to-plane distance (given a distance tolerance) are considered as potential matching candidates. They are then rotated to a plane parallel to a predefined reference one (e.g. the plane Oxy) using a rotation matrix as illustrated in Figure 9. In the next step, the rotated rings are compared based on their shapes, where the numbers or orders of ring vertices do not play a role. Two shapes are geometrically equal if they contain each other's vertices considering the error tolerance ϵ.

4.2.4 MATCHING 3D POLYGON GEOMETRIES

Polygons are extensively used in CityGML as a means to describe surfaces of buildings and building parts. A polygon consists of exactly one exterior and an arbitrary number of interior rings, all of which must lie within the same plane. While an exterior ring defines the outline, interior rings define holes in a polygon (Cox et al. 2004, Gröger et al. 2012, Gröger 2010). Therefore, a polygon can be thought of as a shape bounded by an exterior with all interior rings subtracted from its inner area. The geometric comparison of two polygons is then performed in the same manner as with *LinearRings* in Section 4.2.3.

Note that 3D polygons are matched based on their shapes and not how their individual sub-surfaces are defined. For instance, a polygon with one exterior ring and one interior ring can be matched with another polygon defined by a set of non-intersecting adjacent triangular sub-polygons (i.e. polygon triangulation), if their shapes are geometrically approximately the same. This means that the numbers or orders of sub-polygons contained in the polygons do not play a role, since the implementation merges all pair-wise non-intersecting adjacent sub-polygons, so that they can be represented as a unique shape as a whole. Like rings, polygon shapes can be compared regardless of how their vertices are defined (such as in clockwise or counter-clockwise order with different starting points).

This method has also been extended for the other related objects such as *Surfaces*, *OrientableSurfaces*, *MultiSurfaces* and *CompositeSurfaces*.

4.2.5 MATCHING SOLID GEOMETRIES

A solid is bounded by a set of connected polygons, whose intersections are either empty or an edge shared by both respective polygons. A matching candidate of a given solid can be determined by using its footprint (as a polygon) or minimum bounding box (see Section 4.2.6). However, in contrast to previously discussed geometric entities, matched solid candidates may still be unequal since different solids can have the same footprint or minimum bounding box. Therefore, found candidates are further compared by successively matching their boundary surfaces (i.e. polygons) as described in Section 4.2.4. This also applies to the case where one of the boundary surface is composed of multiple smaller sub-polygons, since the method matches only the shapes of boundary surfaces as a single entity and not how they are formed.

4.2.6 MATCHING THE GEOMETRY OF MINIMUM BOUNDING BOXES

The minimum bounding box of a building is calculated by all its contained geometries, such as ground, wall and roof surfaces (Section 3.4). To make full use of the information available in all dimensions and thus increase the probability of finding correct matching candidates, (3D) minimum bounding boxes are compared based on their shared volume.

Given two arbitrary minimum bounding boxes represented by lower corner points P, R and upper corner points Q, S respectively, their own and shared volume are denoted by V_{PQ}, V_{RS} and V_{shared} respectively. For a given threshold $H \in [0, 1]$, the following applies:

Minimum bounding boxes (P, Q) *and* (R, S) *are matched*

$$\Leftrightarrow \frac{V_{shared}}{V_{PQ}} \geq H \wedge \frac{V_{shared}}{V_{RS}} \geq H .$$

Both ratios $\frac{V_{shared}}{V_{PQ}}$ and $\frac{V_{shared}}{V_{RS}}$ must be compared with the threshold H to exclude the case, where the first minimum bounding box is located completely inside of the second one and vice versa.

5 SPATIAL MATCHING USING AN R-TREE

Section 4.2, particularly Section 4.2.6, determines whether two geometric entities are equivalent and thus can be matched. However, repeatedly comparing all possible pairs of such objects results in a quadratic time complexity $O(n^2)$, which will become a major technical hurdle as the number of city objects in CityGML datasets grows significantly. Thus, to enable more efficient object retrieval and querying, two matching strategies organizing buildings in an R-tree and a grid layout based on their spatial properties are employed in the course of this research, the former of which shall be explained in the following sections. For more details on the grid layout, please refer to Nguyen (2017).

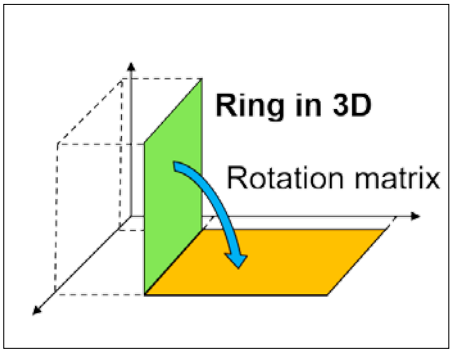


Figure 9: Rotating a 3D (planar) ring

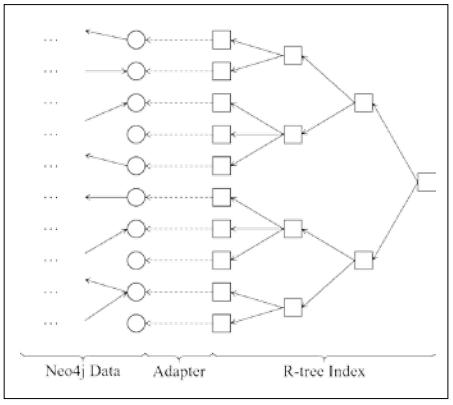


Figure 10: Illustration of an adapter (middle) connecting spatial indices in Neo4j Spatial (right) with data stored in Neo4j (left)

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 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. 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.

5.1.1 CONSTRUCTING THE R-TREE

R-trees are constructed using the plug-in *Neo4j Spatial*. Coordinates of lower and upper corner points of city models are not needed beforehand, since an R-tree automatically expands its envelope on the fly. Building footprints (or 2D minimum bounding rectangles) are however required to construct the R-tree, since the plug-in Neo4j Spatial only supports up to 2D R-trees. In case of unavailable building footprints, they shall be extracted or computed (as described in Section 3.4). Splitting and merging nodes in an R-tree are handled by Neo4j Spatial, which ensures the tree structure is always balanced.

5.1.2 ASSIGNING BUILDINGS TO THE R-TREE

The most important task while expanding an R-tree is to link the spatial information to the data sources it represents. To achieve this, a suitable adapter is needed (see Figure 10), where a connection between an R-tree node and the footprint or minimum bounding box of the respective building is established. Buildings can then be assigned to an R-tree on the fly applying this adapter. Note that each building is assigned to exactly one R-tree node.

5.1.3 MATCHING BUILDINGS USING THE R-TREE

To find the best matching candidates for a given reference building, a query containing its footprint is sent to the R-tree index layer, which then returns a set of R-tree nodes that intersect or overlap with the input footprint. Using the above-mentioned constructed adapter, matching building candidates can be retrieved. If no candidate is found, a delete operation shall be created for the current reference building in the older city model. Otherwise, the best candidate among returned buildings is determined as described in Section 4.2.6. Finally, an insert operation is created for each remaining unmatched building in the newer city model.

The most important advantage of R-trees is the logarithmic time complexity $O(\log_M n)$ on search query operations. Moreover, with the help of Neo4j Spatial, employing an R-tree while matching is simple and straightforward.

6 UPDATING 3D CITY MODELS USING THE GRAPH DATABASE

The matching process in Sections 4 and 5 attaches edit operations to deviation sources on the fly while keeping the actual data untouched. These edit operations can then be converted to WFS transactions and executed accordingly in the updating process in this chapter.

6.1 EDIT OPERATIONS

The general model of all edit operations employed in this research is shown in Figure 11. *EditOperation* is the superclass of all edit operations. It defines a *targetNode*, to which the edit operation is attached, and a flag *isOptional* indicating whether the respective operation must be executed under all circumstances. Such flag is mainly set in scenarios, where geometrically equivalent objects are defined by different syntactic methods. The class *EditPropertyOperation* defines all edit operations created on node properties (i.e. object attributes), while *EditNodeOperation* defines edit operations on the node level (i.e. geo-objects). Figure 12 illustrates how edit operations are attached to deviation sources in the graph database.

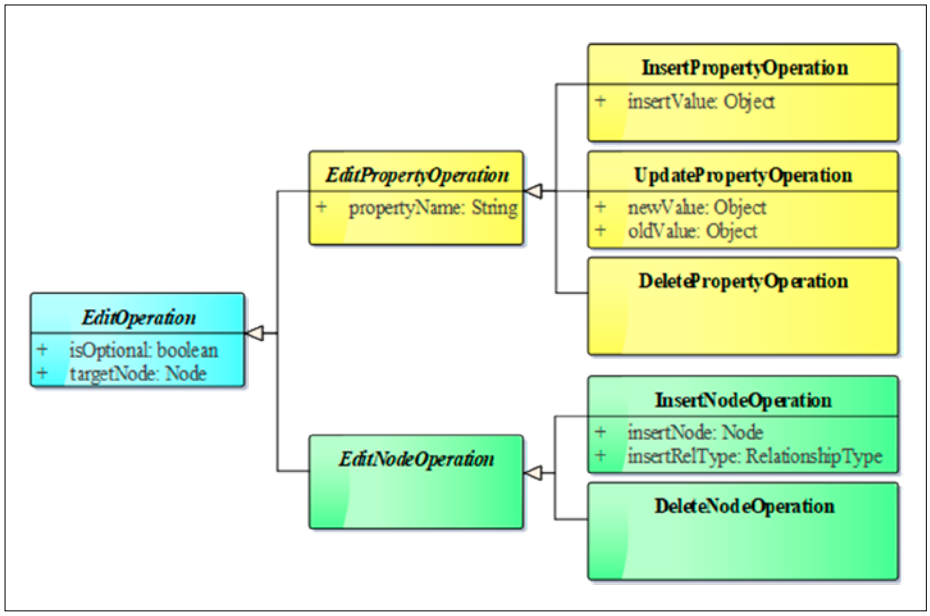


Figure 11: A UML class diagram of all edit operations

6.2 UPDATING BUILDINGS USING THE WEB FEATURE SERVICE (WFS)

EditPropertyOperation objects (i.e. inserts, updates or deletes of node properties) can be transformed to corresponding WFS transactions using their respective stored information. The same applies for *EditNodeOperation* with the only exception of *InsertNodeOperation*, which requires a payload (or content) encoded in XML (Vretanos 2014). XML contents of affected entities can be retrieved by using a Graph-to-CityGML parser, which basically is the reverse of the mapping process introduced in Section 3 as described in Figure 13. For a more comprehensive look at the specifications of WFS requests and responses, please refer to Vretanos (2014) and Nguyen (2017).

7 APPLICATION EXAMPLE

7.1 SCENARIO WORKFLOW

This chapter illustrates a typical use case scenario where the CityGML change detection tool is employed in the context of city modelling and updating involving the 3D City Database. The 3D City Database (3DCityDB) is an Open Source geodatabase for CityGML (3DCityDB 2017). It consists of a relational database schema and several software tools to import, export and visualize virtual 3D city models. As shown in Figure 14, the overall workflow is divided into the following steps:

1. Export the area of an existing city model stored in the relational 3DCityDB, in which updates or modifications should be performed, to a CityGML document using the Importer/Exporter tool.
2. Edit the exported CityGML dataset using some modelling tools (such as the plug-in CityEditor (3DIS 2017) in SketchUp) and save the city model with all the modifications in a new CityGML file.
3. Compare the exported and the edited CityGML document with the help of the change detection tool using a graph database (e. g. Neo4j) as explained in the previous chapters.
4. Update the original city model stored in the 3DCityDB by converting the *EditOperation* nodes created from found deviations to database transactions (such as WFS transactions).

This example illustrates a scenario, in which both the relational 3DCityDB and the graph database Neo4j interact with each other in normal use cases. While Spatial Relational Database Management Systems (SRDBMS) like the 3DCityDB are often used to manage large 3D city models (also in combination with GIS), they are not well suited to matching object-oriented data with multi-level deep hierarchical structure in CityGML. Therefore, by combining a graph database to the existing 3DCityDB and employing the CityGML change detection tool as a connecting bridge, it is possible to edit or update a specific object stored in the 3DCityDB based on the edit operations created in the graph database without having to override the entire database contents.

The first step can be done using the Importer/Exporter tool that comes with the

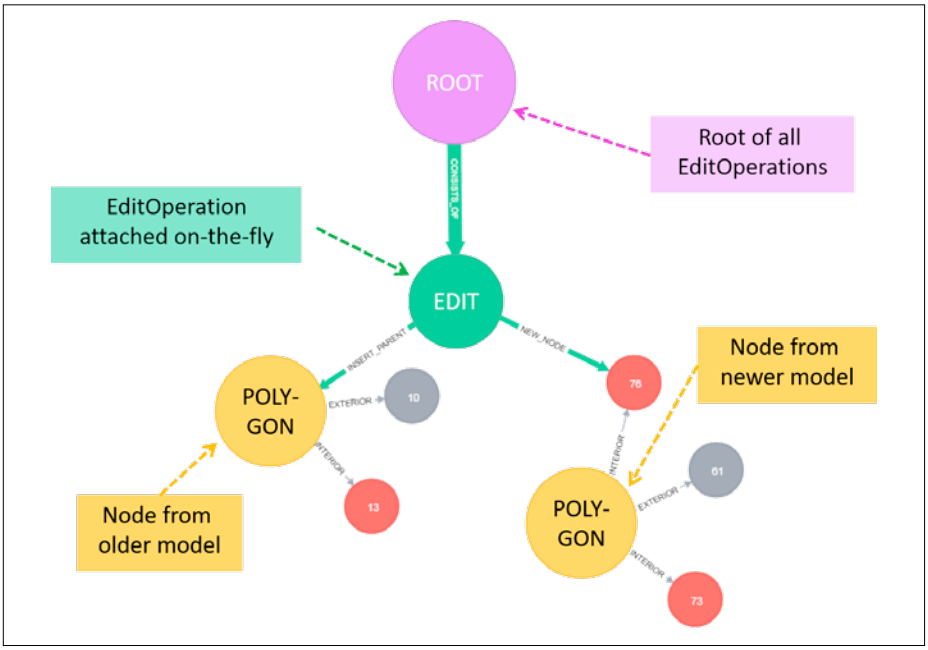


Figure 12: An illustration of how edit operations are attached to deviation sources in the graph database

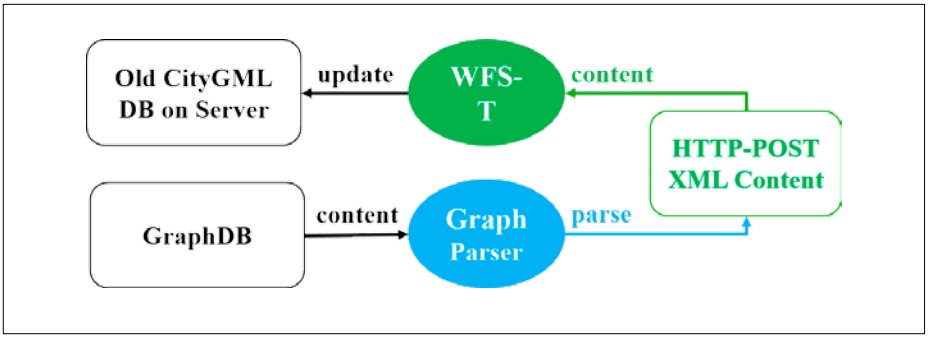


Figure 13: Retrieving XML contents of a CityGML object using a Graph-to-CityGML parser

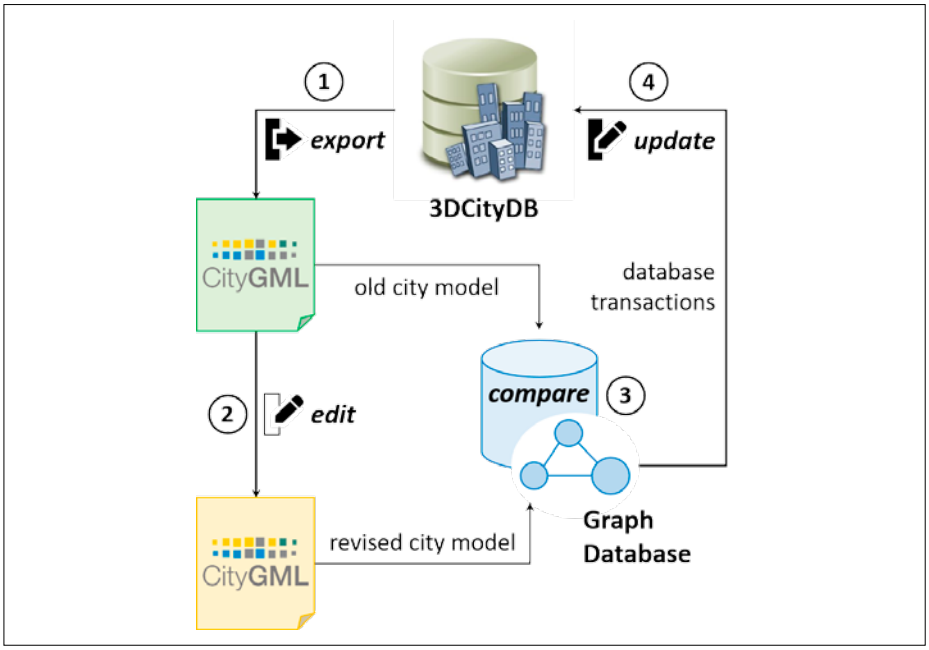


Figure 14: The overall workflow of a typical use case scenario in the context of change detection between CityGML city models stored in the 3DCityDB

3DCityDB, while the remaining steps shall be explained in more details in the next sections.

7.2 EDITING AN EXISTING CITYGML DOCUMENT AND DETECTING CHANGES

This section illustrates an example of how thematic properties and geometric objects of an existing building model exported from a 3D City Model repository (e.g. 3DCityDB) can be modified using the plug-in CityEditor in the modelling software SketchUp. The objective then is to detect changes made by the user to this model. The procedure is described as follows:

1. Import the CityGML model exported from the repository into SketchUp using the plug-in CityEditor (see Figure 15a).
2. Perform some changes to the imported building models. In this simplified scenario, regarding geometrical changes, the upper horizontal edge of one of the walls (marked as green in Figure 15a) is moved upwards along the vertical axis (i.e. *Oy*-axis). The result is illustrated in Figure 15b. On the other hand, regarding thematic as well as attributive information, considering for example the thematic elements in the original document (Box 2).
   The three changes in Box 3 are performed to the thematic data shown in Box 2.
3. Export the modified model to another CityGML file and employ the CityGML change detection tool to find the changes between the two models.

Both the old and the modified CityGML file contain solid geometries bounded by a set of XLinks referring to the composite (boundary) surfaces contained in the respective buildings. Their excerpts are shown in Box 4 and Box 5.

Sometimes, the order of how surface members are listed in the solid objects may change arbitrarily (for example, both of the roof surfaces now appear at the end of the XLink list of the solid object in the modified CityGML file). Since the implementation matches geometrical objects based on their geometry, this does not affect the end matching results, even if the IDs would have been changed.

After the matching process is complete, all found changes are summarized in the console as well as stored in several output CSV files. In this particular test case, a total

Box 2

```
<gen:stringAttribute name="UpdatedBy">
    <gen:value>UserA</gen:value>
</gen:stringAttribute>
<bldg:measuredHeight uom="m">6.947</bldg:measuredHeight>
<bldg:storeysAboveGround>2</bldg:storeysAboveGround>
```

Box 3

Property or Attribute	**Old Value**	**New Value**
uom of *bldg:measuredHeight*	m	urn:adv:uom:m
gen:value of *gen:stringAttribute* "Updatedby"	UserA	UserB
bldg:storeysAboveGround	2	3

Box 4

```
<!– Old CityGML file –>
<gml:CompositeSurface>
   <gml:surfaceMember xlink:href="#Roof_Surface_0_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Roof_Surface_1_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Wall_Surface_0_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Wall_Surface_1_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Wall_Surface_2_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Wall_Surface_3_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Ground_Surface_Poly"></gml:surfaceMember>
</gml:CompositeSurface>
```

Box 5

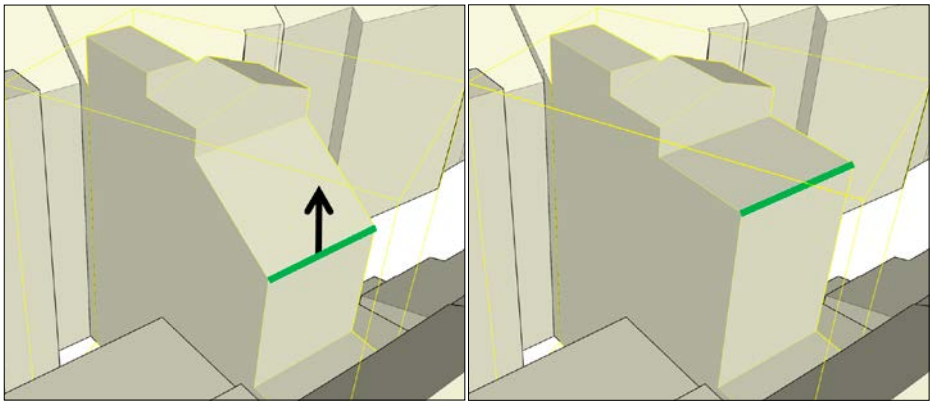
```
<!– Modified CityGML file –>
<gml:CompositeSurface>
   <gml:surfaceMember xlink:href="#Wall_Surface_0_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Wall_Surface_1_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Wall_Surface_2_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Wall_Surface_3_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Ground_Surface_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Roof_Surface_0_Poly"></gml:surfaceMember>
   <gml:surfaceMember xlink:href="#Roof_Surface_1_Poly"></gml:surfaceMember>
</gml:CompositeSurface>
```

Box 6

```
_____________________________________________________
| MATCHER ...                                        |
| Number of UPDATE_PROPERTY nodes:                 7|
| Number of DELETE_NODE nodes:                     0|
| Number of DELETE_PROPERTY nodes:                 0|
| Number of INSERT_NODE nodes:                     0|
| Number of INSERT_PROPERTY nodes:                 0|
|                                                    |
| TOTAL NUMBER OF CREATED NODES:           7 nodes|
| OF WHICH ARE OPTIONAL:                   0 nodes|
| MATCHER'S ELAPSED TIME:                  0 seconds|
_____________________________________________________/
```

of 7 edit operations corresponding to the four changes in geometries and 3 changes in thematic properties are detected and summarized in Box 6.

By moving the green edge in Figure 15a vertically upwards, the three boundary wall surfaces and one boundary roof surface adjacent to this edge are changed during the process. This explains the total number of four geometrical changes found.



a) The 3D model of the selected building before the green edge is moved upwards

b) The 3D model of the selected building after the green edge has been moved

Figure 15: An example of how geometric objects of a building (whose bounding box is highlighted in yellow) imported from the original CityGML dataset can be modified in SketchUp using the plug-in CityEditor

7.3 UPDATING CITYGML OBJECTS STORED IN THE 3DCITYDB

The edit operations created previously are now converted to database transactions required to perform updates in the 3DCityDB. Such transactions can be Structured Query Language (SQL) or WFS transactions, the latter of which can be produced using the tool proposed in this research. However, in order to execute WFS transactions, a transactional WFS (or WFS-T) running on the 3D city model repository (3DCityDB) is required. The commercial WFS version for 3DCityDB provided by virtualcitySYSTEMS is employed in this test case. Box 7 shows an example of a WFS transaction payload required to update the attribute *uom* of the property *measuredHeight* stored in the building *Test_Building* (Box 7).

8 APPLICATION RESULTS

8.1 TEST SETUPS

The following two experiments are performed on a dedicated server-class machine running SUSE Linux Enterprise Server 12 SP1 (64 bit) and equipped with Intel® Xeon® CPU E5-2667 v3 @3.20GHz (16 CPUs + Hyper-threading), a Solid-state Drive Array (SSD) connected via PCIe as well as 1 TB of main memory.

In the first test case, two CityGML datasets recorded at different timestamps of Berlin Moabit (see Figure 16) are compared to each other. Both are encoded in CityGML v2.0.0, contain LOD2 information of approximately 1,100 and 12,300 buildings respectively. Their R-tree footprints are given in Figure 17. The Building objects in these models typically contain a *Solid*, whose *SurfaceMembers* are XLinks referring to existing *BoundarySurfaces*. They also may contain some *BuildingParts* and many Generic Attributes. Note that the matching process can detect changes in all LODs.

The second test case focuses on the tool's performance against massive input datasets. Thus, the entire 3D city model of Berlin (with similar model structure to that of Berlin Moabit) containing approximately 540,000 buildings and occupying 15.5 GB in physical storage is used. The new dataset contains changes added manually to the old one, such as inserts, deletes and updates of the thematic and generic attributes of the objects, as well as inserts and deletes of complex objects such as buildings and their boundary surfaces.

Corresponding edit operations of the detected changes can be found in the graph database as well as in separate CSV files, where all of their information

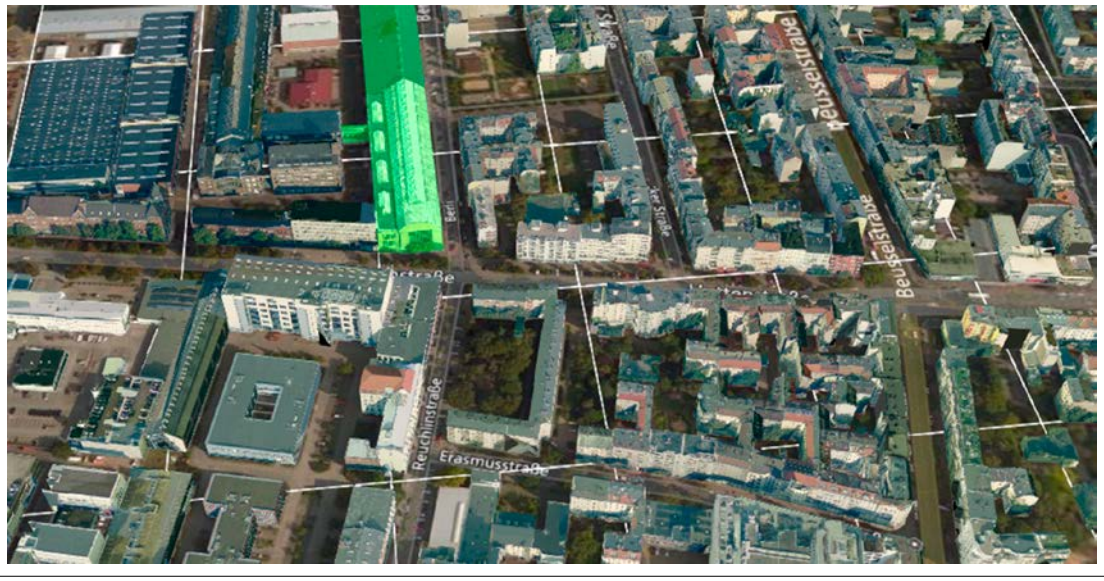


Figure 16: A visualization excerpt of the 3D city model of Berlin (District of Moabit)

Box 7

```
<wfs:Transaction service="WFS" version="2.0.0">
  <wfs:Update typeName="bldg:Building">
    <wfs:Property>
    <wfs:ValueReference>bldg:measuredHeight@gml:uom</wfs:ValueReference>
      <wfs:Value>urn:adv:uom:m</wfs:Value>
    </wfs:Property>
    <fes:Filter>
      <fes:ResourceId rid="Test_Building"/>
    </fes:Filter>
  </wfs:Update>
</wfs:Transaction>
```

Box 8

Whole process	**163 seconds**
Mapping process	98 seconds
Matching process	65 seconds

Box 9

Detected deviations	**170,270**
New properties found	86
Changed property values	95,163
New complex objects found *of which are buildings*	46,915 11,197
Complex objects deleted *of which are buildings*	*28,106* *37*

Box 10

Object Type	**Number of Created Nodes**
Buildings	1,078,364
Building Parts	458
Solids	149,570
Boundary Surfaces	15,407,528
Polygons	12,928,580
Generic String Attributes	23,941,698
Generic Integer Attributes	4,041,104
Generic Double Attributes	3,510,252

(such as source nodes, property names, etc.) is stored. This information can be utilized to create WFS transactions as described in Section 7.3.

8.2 TEST CONFIGURATIONS

Both the testing system and Neo4j share the same Java Virtual Machine (JVM), which is provided with an initial and maximum heap space of 30 GB. The Java concurrent garbage collector *G1GC* is employed. The application configurations for the mapping and matching process are empirically determined to ensure a stable testing environment and optimum throughput. By default (unless specified otherwise), the following configurations are applied: multi-threading with 1 producer and 15 consumers, splitting CityGML elements per collection member (top-level feature), indexing using hash maps stored in memory while mapping, matching buildings using an R-tree with $M = 10$, batch size of 10 buildings and 5000 operations per database transaction.

8.3 EXPERIMENT RESULTS – TEST CASE 1

The test run is performed in under three minutes. The runtime of the mapping and matching process can be found in Box 8.

A total of 170,270 deviations are detected. The deviation types and corresponding numbers are listed in Box 9.

Since the newer city model covers a much larger area than the older model, most buildings (i.e. 11,197 of 12,228 buildings) of the newer model do not exist in the older model and can be thought of as "newly constructed" in the area. On the other hand, 37 of 1,068 buildings of the older model cannot be found in the newer model and can be thought of as "recently abolished". As a result, between both datasets, a total number of 1,031 buildings (11,197 – 12,228 = 1,068 – 37 = 1,031) remain, which means that their spatial position and bounding box do not change over time (although other properties or thematic attributes are changed, as indicated by 86 new properties as well as 95,163 changed properties found). These buildings are all located inside the smaller R-tree footprint of the older city model in Figure 17.

8.4 EXPERIMENT RESULTS – TEST CASE 2

8.4.1 STATISTICS OF MAPPED GRAPH DATABASE

After the mapping process of two 3D city models of Berlin is complete, a total number of 321,142,046 nodes are created (Box 10).

The graph database allocates 126 GB of disk storage in total (excluding the CityGML files and index data).

8.4.2 MULTI-THREADING PERFORMANCE

The multi-threading implementation of the mapping and matching process is realized based on the well-known concurrent producer-consumer design pattern. The differences in performance between various configurations of the numbers of producers and consumers are shown in Figure 18. The results show that the matching process generally benefits greatly from the number of assigned concurrent threads. However, diminishing returns are observed where the total number of producers and consumers

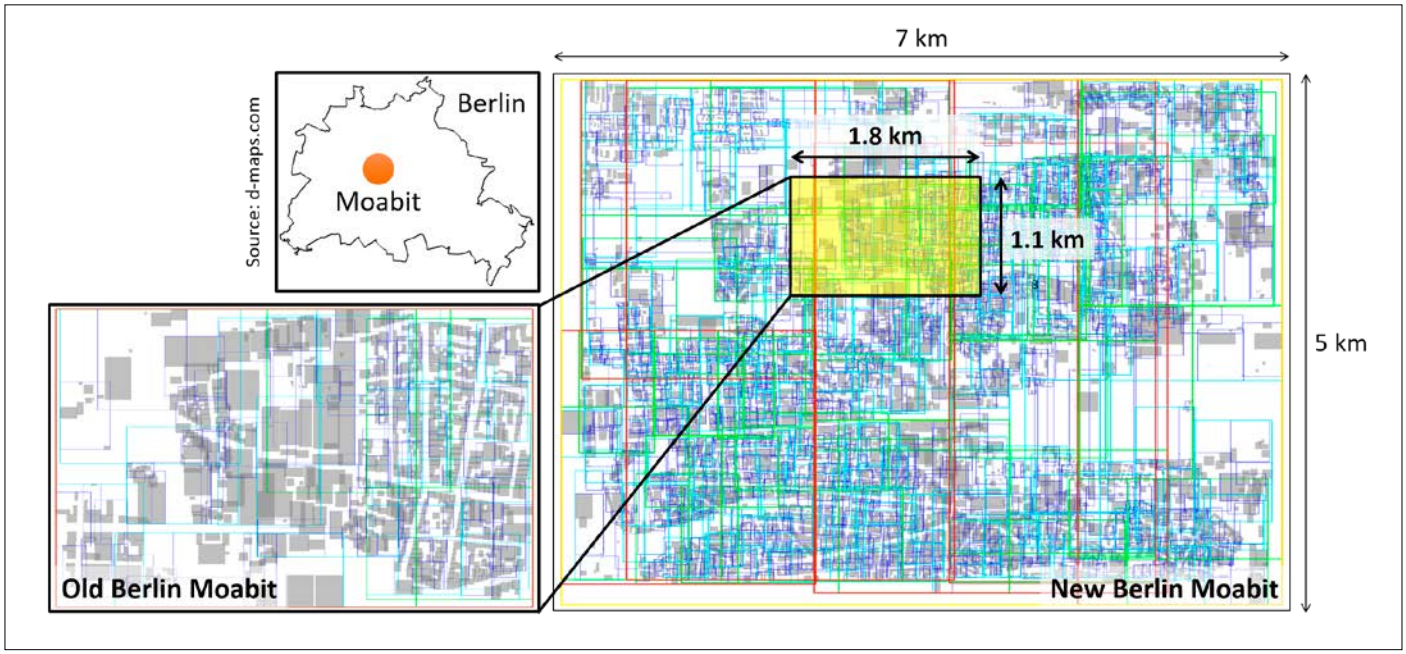


Figure 17: R-tree footprints of the old and new 3D city model of Berlin Moabit. The older model (left) covers a much smaller area compared to the newer one (right).

exceeds that of the testing system's physical CPU cores.

8.4.3 INDEXING PERFORMANCE WHILE RESOLVING XLINKS

Figure 19 shows the significant impact on runtime performance between storing indices in memory (using self-developed internal hash maps) and on disk (using Neo4j legacy indices). The former gives a much better overall runtime performance but requires a large amount of main memory. On the other hand, the latter is more memory efficient but significantly slower due to expensive disk read and write operations.

9 CONCLUSION AND FUTURE WORK

Overall, the mapping process developed in this research is capable of handling arbitrarily large-sized CityGML documents with efficient memory consumption. In addition, it also resolves the syntactic ambiguities allowed in (City)GML between in-line and XLink objects. The matching process can detect deviations between mapped graphs with respect to semantic and geometric properties of city objects. In addition, although LOD2 data were used in the test scenarios, changes in other LODs can also be detected. Moreover, geometric objects such as points, line segments, polygons, surfaces, etc. 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 allocations. These spatial indexing schemes offer a noticeable boost in overall performance. Found deviations are attached to their respective sources in the graph database and transformed to WFS requests complying with the official OGC standards.

The implementation is currently restricted to the modules *Appearance* and *Build-*

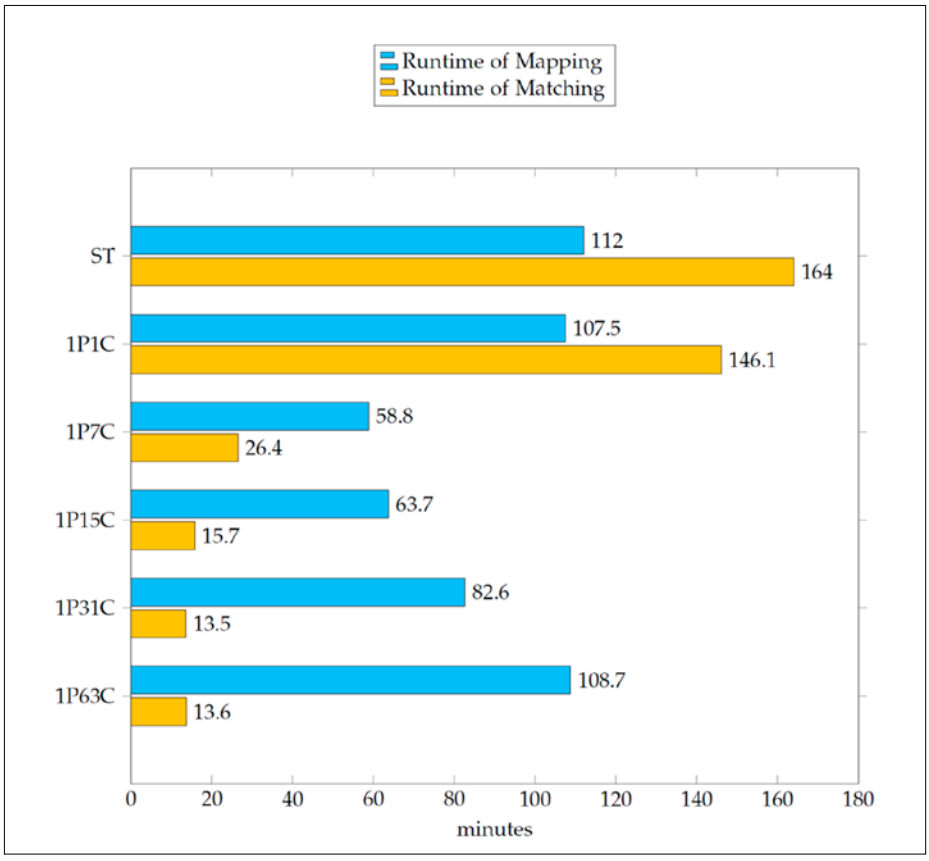


Figure 18: Differences in multi-threading performance. P and C denote the number of producers and consumers respectively

ing (including related objects such as *BuildingPart*, *BoundarySurface*, etc.). Implicit geometries are not included. Moreover, both input CityGML documents must be provided in the same spatial reference system.

In the near future, we intend to extend the implementation to overcome some of the above-mentioned limitations, such as to include additional modules (e.g. *CityFurniture*, *Transportation*, *Bridge*, *Tunnel*, etc.) and to integrate a coordinate system transformation function. Moreover, by utilizing the mapping process, it is possible to perform the analysis of CityGML datasets using their graph representations and graph querying or reasoning tools. In addition, more tests are required to evaluate application outputs against all different types of geometrical deviations. Besides WFS transactions, edit operations stored in the graph database can also be converted to SQL transactions to directly update city objects contained in the 3DCityDB. This is one of the first steps in enabling collaborative work in editing and updating 3D city models in CityGML providing check-out/check-in tools for the 3DCityDB in the future (Chaturvedi et al., 2015). A Graph-to-CityGML and Graph-to-Relational parser are therefore of interest.

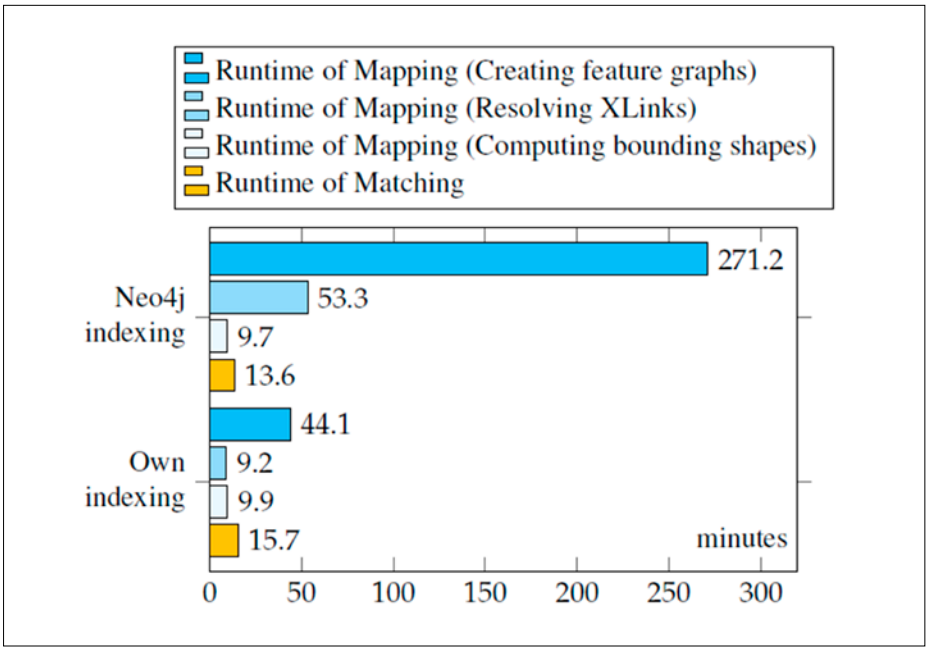


Figure 19: The performance of storing indices on disk (using Neo4j built-in indices) and in memory (using internal hash maps)

IMPRESSUM // PUBLICATION INFORMATION

gis.Science – Die Zeitschrift für Geoinformatik ISSN 1869-9391 // Redaktion: Gerold Olbrich, olbrich@vde-verlag.de, Tel.: +49(0)69-840006-1121 // Hauptschriftleiter: Prof. Dr.-Ing. Ralf Bill, ralf.bill@uni-rostock.de, Tel +49(0)381-498-3200 // Editorial Board: Prof. Dr. Lars Bernard, TU Dresden; Dr. Andreas Donaubauer, TU München; Prof. Dr. Max Egenhofer, University of Maine Orono; Prof. Dr. Manfred Ehlers, Universität Osnabrück; Prof. Dr. Klaus Greve, Universität Bonn; Dr. Stefan Lang, Universität Salzburg; Prof. Dr. Stephan Nebiker, Fachhochschule Nordwestschweiz, Prof. Dr. Josef Strobl, Universität Salzburg // Anzeigen: Katja Hanel, VDE VERLAG GMBH, Telefon +49(0)69/840006-1341, hanel@vde-verlag.de // Anschrift für Zeitschriftenabonnements: Vertriebsunion Meynen GmbH & Co. KG, Cem Küney, Große Hub 10, 63344 Eltville am Rhein, Tel. +49(0)61 23/92 38-234, Fax +49(0)61 23/92 38-244, vde-leserservice@vuservice.de // gis.Science erscheint im: Wichmann Verlag im VDE VERLAG GMBH, Bismarckstraße 33, 10625 Berlin, Tel. +49(0)30/34 80 01–0, Fax +49(0)30/34 80 01-9088, www.wichmann-verlag.de // Geschäftsführung: Dr.-Ing. Stefan Schlegel, Margret Schneider // Verlagsleiter Zeitschriften: Ronald Heinze // Druck: Bosch-Druck GmbH, Ergolding // Erscheinungsweise: 10 x jährlich, davon 4 Ausgaben gis.Science, 6 Ausgaben gis.Business // Jahresabonnement (10 Hefte): 133,00 EUR zzgl. Versandkosten, Studenten/Auszubildende 63,00 EUR zzgl. Versandkosten, Mitglieder des Deutschen Dachverbandes für Geoinformation e.V. (DDGI) erhalten das Abo im Rahmen ihrer Mitgliedschaft // Bezugszeitraum: Ein Abonnement gilt für mindestens ein Jahr und verlängert sich jeweils um weitere 12 Monate, wenn es nicht bis spätestens 6 Wochen vor Ablauf des Bezugszeitraums gekündigt wurde. Bei Nichterscheinen aus technischen Gründen oder höherer Gewalt entsteht kein Anspruch auf Ersatz. // Die gis.Science ist seit 2004 in der internationalen Zitationsdatenbank Scopus gelistet.

Titelbild // Cover image: Das Bild zeigt das Ergebnis einer Solarpotenzialanalyse auf dem 3D-Stadtmodell von Helsinki. Das Besondere ist hierbei, dass zum einen die Globalstrahlung sowohl auf den Dächern als auch auf den Wänden berechnet wurde und zum anderen zur Berücksichtigung der Verschattung durch Vegetation, Dachaufbauten und Balkone das CityGML- sowie das Mesh-basierte 3D-Stadtmodell der Stadt Helsinki gematcht und integriert wurden (Quelle: Lehrstuhl für Geoinformatik, TU München).

References

3DCityDB (2017): The CityGML Database 3DCityDB. https://www.3dcitydb.org, accessed 01/2018.

3DIS – 3D Information Systems (2017): CityEditor – Import, Editing and Export of CityGML Models using SketchUp. https://www.3dis.de/loesungen/3d-stadtmodelle/cityeditor/, accessed 01/2018.

Agoub, A.; Kunde, F.; Kada, M. (2016): Potential of Graph Databases in Representing and Enriching Standardized Geodata. In: Kersten, T. P. (Ed.): Dreiländertagung der SGPF, DGPF und OVG – Lösungen für eine Welt im Wandel. DGPF Publication, Bern, Switzerland, Vol. 25, pp. 208-216.

Bakillah, M.; Bédard, Y.; Mostafavi, M. A.; Brodeur, J. (2009): SIM-NET: A View-Based Semantic Similarity Model for Ad Hoc Networks of Geospatial Databases. In: Transactions in GIS, 13, pp. 417-447.

Berg, M. d.; Cheong, O.; Kreveld, M. v.; Overmars, M. (2008): Computational Geometry: Algorithms and Applications. 3rd Ed. Springer, Berlin/Heidelberg/New York.

Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; Maler, E.; Yergeau, F. (2008): Extensible Markup Language (XML) 1.0 (Fifth edition). https://www.w3.org/TR/xml/, accessed 03/2017.

Chaturvedi, K.; Smyth, C. S.; Gesquière, G.; Kutzner, T.; Kolbe, T. H. (2015): Managing Versions and History Within Semantic 3D City Models for the Next Generation of CityGML. In: Abdul-Rahman, A. (Ed.): Advances in 3D Geoinformation. Springer, Berlin/Heidelberg/New York, pp. 191-206.

Cox, S.; Daisey, P.; Lake, R.; Portele, C.; Whiteside, A. (2004): OpenGIS Geography Markup Language (GML) Implementation Specification Version 3.1.1. OGC document number 03-105r1. http://www.opengeospatial.org/standards/gml, accessed 03/2017.

DeRose, S.; Maler, E.; Orchard, D.; Walsh, N. (2010): XML Linking Language (XLink) Version 1.1. https://www.w3.org/TR/xlink11/, accessed 03/2017.

Egenhofer, M. J.; Franzosa, R. D. (1991): Point-set Topological Spatial Relations. In: International Journal of Geographical Information Systems, 5 (2), pp. 161-174.

Egenhofer, M. J.; Herring, J. (1991): Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases. Technical report, Department of Surveying Engineering, University of Maine.

Falkowski, K.; Ebert, J. (2009): Graph-based Urban Object Model Processing. City Models, Roads and Traffic (CMRT'09): Object Extraction for 3D City Models, Road Databases and Traffic Monitoring-Concepts, Algorithms and Evaluation, Paris, France, 9.

Gröger, G. (2010): Modeling Guide for 3D Objects - Part 1: Basics (Rules for Validating GML Geometries in CityGML). http://en.wiki.quality.sig3d.org/index.php/Modeling, accessed 03/2017.

Gröger, G.; Kolbe, T. H.; Nagel, C.; Häfele, K.-H. (2012): OpenGIS(R) City Geography Markup Language (CityGML) Encoding Standard Version 2.0.0. OGC® document number 12-019. http://www.opengeospatial.org/standards/citygml, accessed 03/2017.

Guttman, A. (1984): 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, MA, USA. ACM, New York, NY, USA, pp. 47-57.

Hunt, J. W.; McIlroy, M. D. (1976): An algorithm for differential file comparison. Computing Science Technical Report 41, Bell Laboratories (1976).

Nagel, C. (2017): citygml4j - The Open Source Java API for CityGML. https://github.com/citygml4j/citygml4j, accessed 03/2017.

Navratil, G.; Bulbul, R.; Frank, A. U. (2010): Maintainable 3D Models of Cities. In: Proceedings of the 15 International Conference on Urban Planning, Regional Development and Information Society. Vienna, Austria. CORP – Competence Center of Urban and Regional Planning, pp. 413-420.

Nguyen, S. H. (2017): Spatio-semantic Comparison of 3D City Models in CityGML using a Graph Database. Master's thesis, Department of Informatics, Technical University of Munich, 2017. https://mediatum.ub.tum.de/1374646, accessed 01/2018.

Nguyen, S. H.; Yao, Z.; Kolbe, T. H. (2017): Spatio-semantic Comparison of Large 3D City Models in CityGML using a Graph Database. In: ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci., IV-4/W5, 2017, pp. 99-106.

Olteanu, A.; Mustière, S.; Ruas, A. (2006): Matching Imperfect Spatial Data. In: Caetano, M.; Painho, M. (Eds.): Proceedings of 7th International Symposium on Spatial Accuracy Assessment in Natural Resources and Environmental Sciences. Lisbon, Portugal, 2006, pp. 694-704.

Oracle Corporation (2015): Java Architecture for XML Binding (JAXB). http://docs.oracle.com/javase/tutorial/jaxb/intro/arch.html, accessed 03/2017.

Redweik, R.; Becker, T. (2015): Change Detection in CityGML Documents. In: Breunig M.; Al-Doori M.; Butwilowski E.; Kuper P.; Benner J.; Haefele K. (Eds.): 3D Geoinformation Science. Lecture Notes in Geoinformation and Cartography. Springer, Cham, pp. 101-121.

Tveite, H. (1999): An Accuracy Assessment Method for Geographical Line Data Sets Based on Buffering. In: International Journal of Geographical Information Science, 13 (1), pp. 27-47.

Vretanos, P. A. (2014): OGC® Web Feature Service 2.0 Interface Standard. OGC® document number 09-025r2. http://www.opengeospatial.org/standards/wfs, accessed 03/2017.

Wang, Y.; DeWitt, D. J.; Cai, J. Y. (2003): X-Diff: An Effective Change Detection Algorithm for XML Documents. In: Proceedings of the International Conference on Data Engineering. Bangalore, India. IEEE Xplore, pp. 519-530.