TUM Department of Civil, Geo and Environmental Engineering
Chair of Computational Modeling and Simulation
Prof. Dr.-Ing. André Borrmann

Investigation of graph-databases for storing and analysing building models

Sining Xu

Masterthesis
for the Master of Science Course Civil Engineering

Author:	Sining Xu
Student ID:	
Supervisor:	Prof. Dr.-Ing. André Borrmann
	M.Sc. Jimmy Abualdenien
Date of Issue:	18. Mai 2018
Date of Submission:	19. November 2018

Abstract

In the architectural, engineering and construction industry, building information modelling (BIM) is playing an increasingly important role in digital representation, intelligent utilization of building models and data exchange between different sectors. Along with the growing size of data and more frequent exchanges, the challenge of BIM lies in determining the reliability of models in different developing phases and their quality before the model is delivered.

Meanwhile in BIM collaboration, IFC data is commonly adopted as a standard format. It contains rich and complex information of building models such as semantic and geometric properties of entities as well as the relationships between them. Therefore, analysing IFC data flaws and weakness of BIM models can provide solutions for realistic BIM challenges.

Concerning data management tool for IFC data, in comparison with the traditional relational database, there have been a proving advantage of graph databases in exploring large interconnected data, which is a main feature of IFC architecture. Thus it will be beneficial to study the potential of graphs to analyse information within BIM. Among graph databases Neo4j is one of the leading systems.

This master thesis firstly illustrates the challenge and theoretical background of the solution. Secondly, detailed methodology of importing IFC data into the Neo4j graph database will be described. This is followed by examples on querying properties of a modelled roof. Furthermore, in combination with Level of Development requirements, property set checks of models in different LODs are performed in order to provide realistic answers in evaluating the reliability of BIM models. As for deliverable models, BIM-based models quality checks about their integrity and physical security are conducted. Through analysing geometric representation in IFC data with spatial operation, consistency and intersection detections are executed. These queries demonstrate that graphs can efficiently manage IFC data and provide chances to build diverse applications.

Overall this study presents practical syntaxes in analysing IFC data within Neo4j database and investigates advantages and limitations of this mechanism. Finally, the work is finished by a conclusion and possible solutions to overcome these limitations.

Contents

List of Abbreviations

2D	Two-dimensional
3D	Three-dimensional
AEC	Architecture, Engineering and Construction
AECO	Architectural, Engineering, Construction and Owner/Operator
AIA	American Institute of Architects
APOC	Awesome Procedures On Cypher
BIM	Building Information Modeling
BRep	Boundary Representation
CSG	Constructive Solid Geometry
IDE	Intergrated Development Environment
IDM	Information Delivery Manual
IFC	Industry Foundation Class
ISO	International Organization for Standardization
LOD	Level of Developmemt
MEP	Mechanical, Electrical and Plumbing
MVD	Model View Definitions
NoSQL	Not Only Structural Query Language
SMC	Solibri Model Checker
SQL	Structural Query Language
STEP	Standard for the Exchange of Product model data

Typographical Conventions

The following typographical conventions are used in this thesis:

Italic to indicate the IFC classes and Python commands within the text

UPPER CASE and Italic to indicate Cypher commands within the text.

Monospaced (“typewriter”) family is used for program listings to indicate Cypher commands.

List of Figures

List of Tables

Code Listings

Chapter 1

Introduction

1.1 Introduction and Problem Description

Traditionally construction plans are represented by 2D-drawings and models, which are still widely used today. With the help of computer technology, digitalization of buildings have become much more convenient. Meanwhile construction projects are also becoming more complex and requiring more detailed and interdisciplinary planning. Therefore a more efficient management system of construction data is of wide interest. In the new century the concept of Building Information Modelling was developed. It is featured by fully integrated, interoperable digital information which can be used by all members of design, construction and operation team throughout the facility's life cycle(Holness, 2006). Nowadays in architecture, engineering and construction industry, building information modelling (BIM) is playing a more and more important role in digital representation of constructions and data exchange between different sectors.

Along with the development of BIM comes the challenge that different design disciplines and project organizations, such as architects, engineers, owners and construction crew, require different information to be available at different project milestones(Treldal *et al.*, 2016). Moreover, clarification of data about to what extent could the model be relied on and from whom can it be authored will also facilitate the information exchange between various teams. To describe the content and reliability of building elements during the design and construction process, many organizations have developed standards about the definition of modelled objects and information embedded within them. Among them is the Level of Development Specification (LOD) released by BIMForum based on the AIA(American Institute of Architectures) protocols, one of the most important points of reference of several BIM guidelines(Bolpagni, 2016). It intends to improve the quality of communication among users of BIM about the characteristics of elements in models(BIMForum, 2017). Level of Development Specification consists of a series of definitions about the content, which an element should include at various stages in the design and construction process. These definitions present not only the geometrical representations of elements but also their association to semantic characteristics.

A model can include elements at different levels of development owing to the different rate of progression. In other words, a model could hardly be defined by one LOD. Additionally, there is no unified standards for the design phase. One LOD can have different requirements between different participants in the AEC industry. Therefore, in a complex model which contains a huge amount of information and relationships, it is often demanding to check and compare the level of development of all the elements and entities for different teams.

Apart from checking data content and existence during the developing phase, it is also important to analyses deliverable end models for downstream utilizations for integrity, quality, and physical security, since BIM models consist of an amalgamation of models from different disciplines which may work in tandem. These models can contain flaws and weakness despite meeting the requirements of certain LOD. For instance the slabs may overlap and columns may not be continuous. These problems will negatively affect reliability of models and thus impede data exchange between different design teams.

Comparing with traditional design method, it is much more convenient to spot flaws using BIM-based model checking. BIM-based model checking (BMC) can be regarded as one of the best ways to illustrate the benefit of the relevant content of information in BIM-files(Hjelseth, 2015). One of the most important advantages is that numerous rules can be applied. Among them, clash detections and deficiency detections are key issues in some commercial checking software. Thus, conducting clash detections and deficiency detections in Neo4j is important in facilitating BIM development.

Meanwhile, as a technical basis the Industry Foundation Classes (IFC) is commonly adopted as a standard format for collaboration between BIM contractors and data exchange between different modelling softwares. The IFC data defines a rich and complex object model, including the properties of them, such as geometric properties and non-geometric properties, as well as the relationships. It contains the necessary information to define level of development and conduct model checking. But exploration of these information requires a deep understanding of the IFC object model, efficient data management and query tools needs to be introduced(Ismail *et al.*, 2017).

How to efficiently manage data in IFC format has been a frequently researched topic. Traditionally relational database and associated structured query language (SQL) is one of the most frequent combinations for intensive data storage and retrieval applications(Vicknair *et al.*, 2010). However, within IFC data there are not only geometric and semantic informations, but also numerous relationships between entities and elements. Moreover, when it comes to comparison of models within different LODs, it is necessary to match and compare them through comparitive relationships. In contrast with the more recent graph databases and its associated NoSQL languages, relational database is not as efficient in processing data with a lot of connections(Vicknair *et al.*, 2010). In many different domains the ability of graph database in processing complicated interconnected datasets have been proved. Within BIM sector, the relationships and properties between different building entities can be more directly and efficiently represented using graph data. "Hence converting of BIM models based on the IFC standard into an effective information retrievable model based on graph databases could significantly facilitate the efforts of exploring and analysing BIM highly connected data"(Ismail *et al.*, 2017). Therefore IFC query systems based on graph data and its application has become the focus of considerable research efforts in recent years. In conclusion the focal point of this assignment is:

Investigation of graph-databases for storing and analysing building models.

1.2 Aims and Objectives

To allow management of BIM information in graph models, the first step is to transfer IFC data model into a graph database. Secondly, several fundamental property queries on a building element, which is the roof in this study, will be executed to explore the concept of graphs in analysing IFC models. Then, more complex property sets based on certain LOD requirements will be analysed in order to evaluate the reliability of models in different phases.

As for deliverable end models, the application of graph databases in assessing their quality and integrity is also explored. Based on spatial queries on IFC geometry representations, several problems are detected and analysed. These problems include deficiency checks and intersection detection, which are realistic in the AEC industry.

1.3 Layout of the Thesis

Chapter 1 describes a potential data management problem in BIM industry and presents a possible solution by analysing IFC data in graph databases.
Chapter 2 introduces the theoretical background of BIM and its important concept: level of development. This is followed by IFC architecture and the graph database which will be used to analyse it. Roof components are also discussed as a background of the roof model.
Chapter 3 presents the research approach and lists used tools. In this chapter the detailed methodology of importing IFC data into graph database Neo4j is illustrated, which lays a foundation for further researches.
Chapter 4 demonstrates the property query capability of graphs. Furthermore, after

analysing relevant IFC data structure, queries based on realistic LOD requirements on properties are conducted to evaluate reliability of models in different design phases.
Chapter 5 discusses applications of graphs in model checking. Through exploration of IFC geometry representations, queries about realistic problems, like consistency check and intersection detection, are conducted.
Chapter 6 concludes the advantages and limitations of graph database. In addition, future research directions and recommendations are addressed to provide a solution to the problems and improve cooperation between graph databases and BIM models.

Chapter 2

Theoretical Background

This paragraph illustrates the theoretical basis for studying into the research goal. To lay the foundation, concept of Building Information Modelling will be explained in the first section. Together comes an important part of BIM, the maturity of information which is mostly referred to as level of development (LOD). Definitions and meanings of LOD will be discussed to clarify the important aspects describing degrees of completion. Afterwards will the research tool graoh dabatase and non-SQL database, in comparison with traditional relational database and SQL language, be illustrated and the advantages of the graph database Neo4j be explained. Moreover, the structure of IFC format, which is used for caring and analysing model data is discussed. In order to interpret IFC data, IFC libraries and parsers are used, which will be covered in this section.

2.1 BIM and Level of Development

2.1.1 Building Information Modeling

BIM stands for Building Information Modelling in the architecture, engineering, and construction (AEC) Industry. It is not only about digital representation of physical and functional characteristics of buildings, but more about intelligently using the digital model through the whole service life of a structure - from design, construction, operation to demolition.

2.1.2 Level of Development

In order to effectively deliver building projects and check the reliability of entities, it is essential to describe what information is needed, from whom, and at what level of detail(Bolpagni, 2016). Among these definitions are the semantic specification and geometry specification widely considered as important parts of classifications system about the maturity of model content.

Many institutions and associates have established specifications about the information which should be embedded in modelled elements to establish development milestone definitions. Between them, the Level of Development Specification published by the BIMForum is one of the most adopted guidelines and documents. The BIMForum's interpretation of the LODs are defined as LOD100, LOD200, LOD300, LOD350, LOD400. LOD500 is taken as part of field verification. The increasing number indicates more detailed and precise element.

Level of development describes the maturity of the model content and lays a foundation for BIM contracts. A precise definition of LOD will reduce the risk of miscommunication among members of project teams(BIMForum, 2017).

2.2 Comparison of Databases

2.2.1 Relational Database

Traditionally in most data management and retrieval applications, relational databases have been the primary storage structure. Relational database organizes data into tables which are defined by sets of rows and columns. Each table represents one entity type while columns stands for attributes of them and rows can be perceived as instances of that type(Sarwar *et al.*, 2001). Between tables can there be logical connections between them. The connections are defined through indicating the unique primary keys of relevant tables. More exactly, one-to-many relationship is realized by migrating own key into foreign tables and many-to-many relationships are demonstrated through creating additional key tables that contain the primary keys from both of the other entities. Retrievals in relational database are usually accomplished using structural query languages (SQL).

One of the limitations of relational models is that if the data contains large amount of relationships, it requires huge joins of large tables. These big data problems like modelling social network, bioinformatics calculation and BIM model data management are becoming increasingly common in science and industry today. Storing, retrieving, and manipulating such complex data becomes onerous when using traditional relational database system approaches(Miller, 2013), because of the frequent misleading migration of keys and numerous key tables. With the increasing demand in storing large amount of interconnected data, new storage alternatives to relational database are being developed. These new systems are categorized as NoSQL systems, in which graphs are one of the most for interconnected data optimized database(Batra & Tyagi, 2012).

2.2.2 Graph Database and its Advantages

Graph database uses graph structure, which is composed of vertices, edges and their relevant properties, to represent and store data. Nodes stands for objects and edges manifest relationships between nodes. Both nodes and edges can have properties. Among graph databases is Neo4j one of the most leading storage systems.

Many studies have shown that graph databases are effective for processing dense, interrelated datasets because of graphs' emphasis on exploring relationships between entities. The design of graph structure allows navigating and filtering through correlations and patterns. Moreover, the highly dynamic data model in which all nodes are connected by relations allows for fast traversals along the edges between vertices(Miller, 2013). Since relationships and properties also play an important part in IFC data structure, it is beneficial to make use of the advantage of graphs to explore IFC data in BIM.

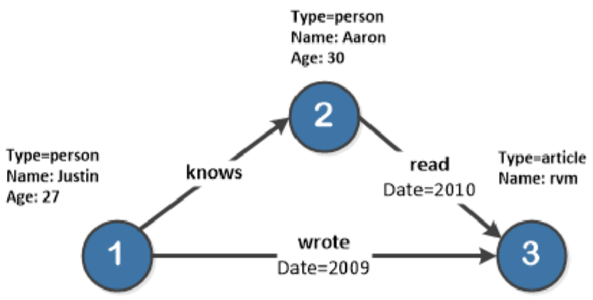


Figure 2.1: Illustration of Graph Database Structure(Miller, 2013)

2.2.3 Neo4j Graph Database

Neo4j is a NoSQL graph database management system released in 2010. It is implemented in Java and uses Cypher query Language. It is one of the world's leading graph database with native graph storage and processing. Neo4j has the following advantages against relational database[1]:

1. faster transactions and processing for data relationships.
2. flexible, data types and sources can be added or changed at any time.
3. processing performance is regardless of the number or depth of relationships.

In Neo4j, data is stored in the form of either an edge, a node, or an attribute. Nodes can have any number of properties and labels. Beside,s they can be connected to any number of other nodes through relationships. Relationships are directional and also have one or more properties. It is possible to index or constrain their properties. Moreover, labels are used to group nodes into sets which work as a filter and indicator to narrow research. Graph analysis

[1]https://neo4j.com/product/

can be performed in Neo4j. The input data as well as the result can be visualised as graphs to offer new perspective of the data or build intelligent applications.

2.2.4 Cypher Query Language

During the development of NoSQL graph database Neo4j, its supporting query language, Cypher is invented. Cypher is a declarative graph query language that allows for expressive and efficient querying as well as updating of the graph. It enables users to directly state what to select, insert, update or delete from the graph data without having to know the exact procedure. Some key words and expressions in Cypher are inspired from SQL and other declarative query language for querying graph data(Wikipedia contributors, 2018a).

Nodes and relationships are the building blocks of Cypher graph model. Nodes have labels and properties on them while relationships can contain type information. Pattern, which works as an important navigator and filter in Cypher, consists of multiple nodes and relationships.

The most used clauses in Cypher are *MATCH*, *WHERE* and *RETURN*. *MATCH* is used to get desired data from the graph by describing the pattern or properties. *WHERE* can add constraints to the pattern or filter results passing through *WITH*. *RETURN* returns the queried result. In addition, *WITH* can manipulate the output before it is passed on to the next sector. These four clauses play an important part in the Syntaxes mentioned below.

2.3 IFC Standard

The Industry Foundation Classes(IFC) is an open standard for the exchange of building data models used in AEC industry across different software to improve collaboration of the building project. IFC is based on the ISO-Standard STEP (ISO 10303), which stands for standard for the exchange of product model data. IFC has inherited data modelling language EXPRESS and the exchange of a model via STEP physical file and geometric modelling from STEP(Borrmann, 2016). EXPRESS language also specified a graphical representation known as EXPRESS-G to provide graphical subset. It was developed and maintained by buildingSMART International and is an official standard ISO 16739. Not only within a project team, but also between software applications used in design, construction, procurement, maintenance and operation, it can be used to exchange information(Autodesk, 2018). IFC is already a mandatory format for all public building projects in most of the Scandinavia countries and is being recognized by more and more countries.

2.3.1 IFC Data Model

IFC architecture standardizes data structure for the exchange of building information models including geometry and semantics, which the IFC model view definitions mainly supports. IFC format stores models with object-oriented data and objectified relationships. It has a large inheritance hierarchy(Borrmann, 2016).

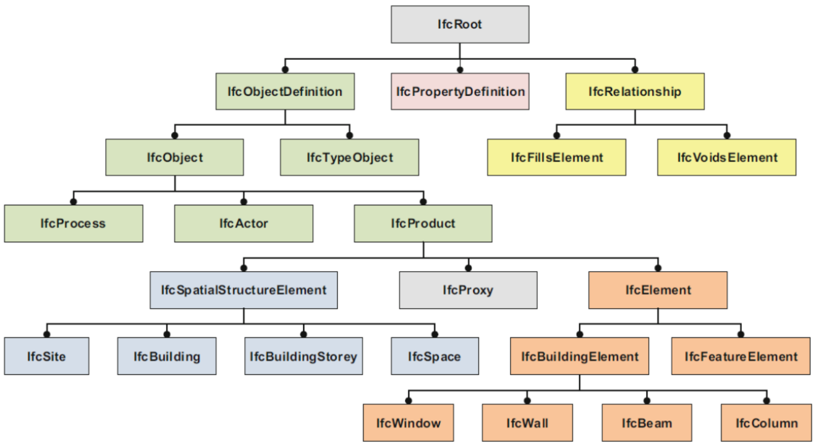


Figure 2.2: IFC Inheritance Hierarchy (Borrmann, 2016)

As we can observe from the graph, the IFC data structure has three fundamental entity types: *IfcObjectDefinition*, *IfcPropertyDefinition* and *IfcRelationship*(Ismail *et al.*, 2017). *IfcPropertyDefinition* describes all the characteristics attached to objects. For instance, the information about fire rating of an element is stored in class *IfcPropertySingleValue*, which is an important type when doing queries on properties and its associate object. *IfcObjectDefinition* represents modelled objects or process like *IfcRoof*, *IfcSlab* and *IfcWindow*. *IfcRelationship* stands for the objectified relationship between different classes, including relationships between objects, relationships between properties and relationships between properties and objects. Relationships can have information attached to them too. By retrieving the item tags indicated in the relationship node can relationships between the tagged entities be found. Because of this object-orientation, it is useful when finding connections and paths between desired nodes in the queries below. Figure 2.3 demonstrates different types of objectified relationships and their abstract super-type *IfcRelationship*.

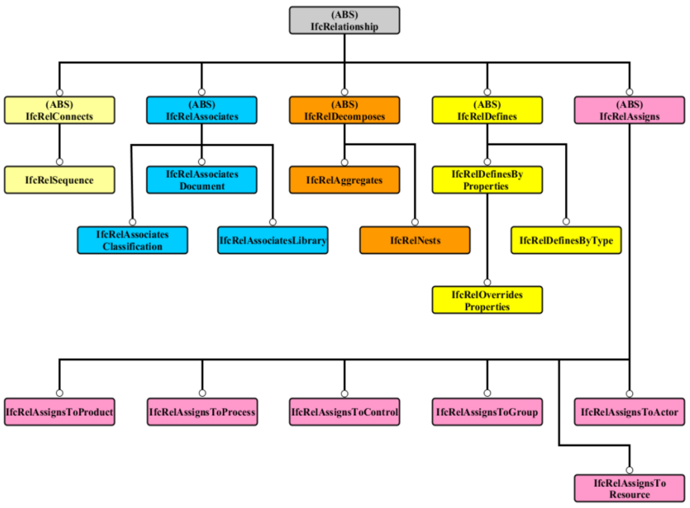


Figure 2.3: IFC abstract objectified Relationship (Ismail *et al.*, 2017)

Besides, IFC classes can have direct attributes attached to them which could indicate a relationship to another object or they could just be attached as simple data type attribute, e.g. integer, string, logical, or Boolean(Ismail *et al.*, 2017). With the help of these indicators can IFC classes inherit properties from or pass on them to other nodes without repeating them in all the nodes.

2.3.2 Model Views

Data exchange using IFC can lead to problems. Because of the difference of geometry representations between different software, vagueness of implementation or errors from software vendors or users can occur. To solve this, a subset of full IFC schema that defines the desired objects and attributes is introduced. These subsets are called model views. In other words, Model views are used for the targeted exchange of specialized models.

In Revit there are some predefined views like IFC 2×2, IFC 2×3 and IFC 4. Among them is IFC 2×3 Coordination View Version 2.0 currently the most widely used and supported model view definition. It supports the rudimentary parametric derivation of building components when importing into planning tools(Autodesk, 2018). Therefore, this model view will be adopted when exporting IFC files.

2.3.3 Geometric Representation

A 3D model is the most complete representation of the nominal shape of a model. In the IFC $2\times$ model, three different types of solid model representations are defined(Liebich, 2009):

1. Swept area solid representation;
2. Boundary model representation;
3. Constructive solid geometry (CSG) representation;

Swept area solid representation is the most common and simple graphical method. It will be adopted when the form can be represented by a defined profile that is led along a path, which might rotate or distort, to generate the solid. Take roofs as an example, they will be indicated by *IfcShapeRepresentation* with *RepresentationIdentifier* as "Body" and *RepresentationType* as "SweptSolid". The geometry is described by *IfcExtrudedAreaSolid*, which is usually applied for standard slabs).

```
#151= IFCEXTRUDEDAREASOLID(#147,#150,#19,0.352);
......
#161= IFCSHAPEREPRESENTATION(#97,'Body','SweptSolid',(#151));
```

Code Listing 2.1: IFC Expression of "SweptSolid" Shape Representation

For example, node 161 shows that the geometry is represented by extrusion. It is linked to node 151 *IfcExtrudedAreaSolid*. Node 151 shows the thickness (0.352) and is connected to its position (#150, #19) and profile (#147). Therefore it is direct and convenient to retrieve the details of a swept solid by querying through this pattern. If a slab has openings or recesses and trenches, they are exchanged using *IfcOpeningElement*, assigned to the slab through the *IfcRelVoidsElement* and accessibly from the *IfcSlab* entity through the *HasOpenings*(Liebich, 2009). The *IfcSweptAreaSolid*, when referencing parametrized profile definitions, is normally used as the preferred geometric representation for building elements(Liebich, 2009), when the shape is not complicated.

Brep stands for boundary representation and can also be described as boundary surface model. The basic components are faces, edges and vertices. The surfaces of a component are represented using coordinates and lines. They form the actual solid together, allowing even complex forms which cannot be generated with swept solids and extrusions(Autodesk, 2018), such as pitched roofs with openings or windows. Normally it is used to describe the explicit form of the object. The type is indicated by *IfcShapeRepresentation* with *RepresentationIdentifier* as "Body" and *RepresentationType* as "Brep".

```
#3330= IFCFACETEDBREP(#3328);
......
#3340= IFCSHAPEREPRESENTATION(#97,'Body','Brep',(#3330))
```

Code Listing 2.2: IFC Expression of "Brep" Shape Representation

The representation type is given by node 3340 and it points to the actual surface B-rep node 3328, which further links to boundary surfaces and vertices.

Because of its complex calculations to represent surface details, it uses more data memory. Moreover it is harder to comprehend the shape when it is exported into another software, for instance when parsed by the parser IfcOpenShell in python. Extra software like pythonOCC and OpenCscade BRep needs to be installed. Therefore in order to avoid complex queries, it is beneficial to use representation type swept solid. The roof is modelled as simple flat roof so that it can be represented by swept solid.

Constructive solid geometry representation is an implicit geometry representation. It is based on a collection of primitive objects that are combined by Boolean operations. The use of CSG is currently restricted to the Boolean operations on other solid models, as no CSG primitives are included in the IFC2x specification(Liebich, 2009).

2.3.4 Semantic Representation

Geometry information and semantics information are strictly separated in IFC. Semantics information include attributes and parameters defining an object. It is structurally organized and well broken down into classes. Property definitions can be either(Liebich, 2009):

1. type defined and shared among multiple instances of a class;
2. type defined but specific for a single instance of a class;
3. extended definitions that are added by the end users;

For object occurrences which share a same set of properties, they are assigned an *IfcTypeObject* through relationship *IfcRelDefiniesByType*. For example, the property of a window (*IfcWindowStyle* and *IfcWindow*) is assigned by node 815 *IfcRelDefinesByType* by indicating their node ID 715 and 753.

```
#715=IFCWINDOWSTYLE('1gQnfh5u1BRuzf3uNuBOQa',#41,'100 x 150',
$,$,(#714),(#712),'395915',.NOTDEFINED.,.NOTDEFINED.,.F.,.F.);
......
#753=IFCWINDOW('0WenMeXu5D7xxOqpsCw2De',#41,
'Dachfenster 1-flg:100 x 150:395932',$,'100 x 150',#752,#747,
'395932',0.454279315745163,1.06000000000008);
......
#815=IFCRELDEFINESBYTYPE('3_HzOq$Z56aeMNTlNSsfWi',#41,$,$,
(#753),#715); #41,$,$,(#177),#307);
```

Code Listing 2.3: IFC Expression of Window and its shared Property Set through *IfcRelDefinesByType*

To describe multiple instances of af a class will *IfcPropertySet* be used. These properties are assigned through *IfcRelDefinesByProperties* relationship. Same instance of *IfcPropertySet* can be attached to all objects within a type. The difference from *IfcTypeObject* is that the property sets can differ from each other in values within a private copy.

```
#177=IFCSLAB('2D2_zwByDCRR4wSXeznz$x',#41,
'Basisdach:Roof LOD 300:391113:1',$,
'Basisdach:Roof LOD 300:391764',#176,#164,'391113',.ROOF.);
......
#307=IFCPROPERTYSET('2D2_zwByDCRR4wUUCznz$x',#41,
'Pset_SlabCommon',$,(#295,#296,#304,#305,#306));
......
#313=IFCRELDEFINESBYPROPERTIES('1ho$$qbS5D$gWsyrRfWJnm',
#41,$,$,(#177),#307);
```

Code Listing 2.4: IFC Expression of Slab and its private Property Set through *IfcRelDefinesByProperties*

For example, the slab property #177 is specified by node 313 *IfcRelDefinesByProperties* to #307 *IfcPropertySet*. *IfcPropertySet* then points to the relevant *IfcPropertySingleValue* that defines its attributes and parameters in detail.

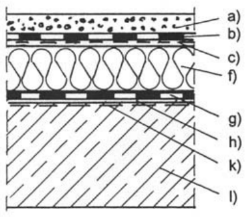
2.4 Roof Level of Development Definition

To compare Roofs in different stages, firstly a LOD standard or threshold to describe the difference should be developed. Comparisons between geometrical data like thickness, height and semantic data like materials, functions are quite different. For geometrical dimensions, numerical difference or standard deviation can function as a guideline. In contrast they do not work for semantic information. Therefore a requirement specification is needed.

As a result of various designs and accessories of roofs, before modelling a roof for definition and comparison between roofs with different levels of development, first we need to decide which functional and physical characteristics are important for it.

2.4.1 Types of Roof Build Ups

According to different layer sequence and constructions above the supporting structure, there are generally three types of flat roofs: warm roof, cold roof and inverted roof(Winter, 2016).



Prinzipaufbau eines **Warmdachs**:
a) Oberfläche (Schutz, Auflast, Nutzschicht)
b) Dachabdichtung
c) Dampfdruckausgleichsebene
f) Dämmung
g, h, k) Dampfsperre auf Ausgleichsschicht und Voranstrich
l) Geschossdecke
Quelle: Flachdachrichtlinien

Figure 2.4: Build up warm roof (Winter, 2016)

Among them is warm roof one of the most widely constructed roof build ups which can be applied on most supporting structures made of different material types like concrete, timber, profiled metal, or in different forms like pitched and flat roof. The main principle is that the thermal insulation layer is located above the structural decking and under the waterproofing. In order to prevent the moisture vapour hidden in the roof build up being forced into insulation through outside sun heating or inside thermal pressure, a vapour incorporate layer under insulation is essential. Another necessary part is the separation of insulation and waterproofing coating due to the different coefficient of linear expansion of them, so as to prevent the decoupling of these layers. Above the waterproofing comes eventually a surface protection like tiles or ballast to provide protection against weather or loads. It is easier to eliminate roof void ventilation and cold bridging in comparison to cold roof since it has a continuous layout from structure to the outside. Moreover it is more stable and easier to build not only on flat roof but also on profiled decking or pitched roof.

Cold roofing normally comprise of a structural deck with a ceiling layer fixed to the bottom of joist thereby creating void within. Insulation is normally placed above the structure, in the lower part of this cavity, while the waterproofing on the outside of the deck. Therefore the outside structural deck and coating is not heated by the building and stay in lower temperature. Therefore when the warm air on the room side meets the cold air on the deck side the water will possibly condensate. Vapor control layer above ceiling level or even ventilation must be provided to reduce the risk of interstitial condensation. Another problem is that it is harder to prevent cold bridging which is formed by the joist or purlins between outside roof deck and inside structure.

Owing to the problem with roof void condensation and cold bridging between inside and outside, cold roof is now seldom constructed. Therefore it will not be considered as an example.

Another widely used roof build up is inverted roof, which was invented in USA in the 1950s. The main feature is that the principal thermal insulation layer is located not only above structure but more importantly above waterproofing. Strictly speaking it can be taken as warm roof since the layers are heated by the building. The waterproofing is thereby protected by upper layers from effects of weather and therefore more robust. On the other hand since the insulation is loose laid on waterproofing and exposed directly to the outside, a protection layer against particles and ballasting against uplift is needed. But one of the disadvantage is that because of the imposed loading of the protect layer, it is generally only built on flat roofs, which limits its potential and application.

In conclusion, because of its wide application and clear, continuous construction, warm roof will be taken as the modelled sample.

2.4.2 Components of Warm Roof

2.4.2.1 Practical Roof Build Ups from different organizations

Theoretically warm roof is composed of waterproofing, insulation, membrane layer and vapour control layer. But how they are practically represented in Revit can be different. From the NBS(National Building Specification) national building information modelling library, which is one of the fastest-growing BIM library in the UK with objects meeting the requirements of internationally-recognised NBS BIM object standard(Nationalbimlibrary.com, 2018), warm roof models from five different organizations are downloaded and compared so as to reveal their digital representation. These models are Kemperol V210 warm roof system (STRATEX) fully re-enforced liquid applied waterproofing system from Kemper System Ltd, Thermo-planFPO warm roof system from Bauder.Ltd, compact roof systems with with membranes and timber decking from FOAMGLAS building, pitched aluminium sheet roof on timber trussed rafters from NBS generic roofs, Elastaseal^{TM} Warm Roof System 20 liquid-applied reinforced polyurethane membrane system from Tor Coatings Ltd.

2.4.2.2 Summary Practical Roof Build Ups in Revit

According to their layouts given in product descriptions, since the build ups are organized in various ways, it is necessary to summarize them with one standard. Firstly we need to know how roofs are assembled in Revit. In Revit are roofs categorized as finish, structure, thermal/air layer, membrane layer and substrate. These layers are possible to be modelled more than once. Vice versa, sometimes will one group contain more than one layer.

Finish refers to outside skin of the roof like tiles, which generally servers to protect the layer under it.

The definition of structure is relative vague. In Revit normally will joist and boards simply regarded as structure. In this thesis, structure will be separate into two parts – structure joist/battens and structure supporting decks, because different roof build ups may contain either one or both of them.

For some companies bitumen sheets are also taken as structure, or even finish, but here it will distinguish from structure and set as membrane layer, which indicates sheets, barriers or coatings. These elements can be more detailly modelled as waterproofing, vapor control layer and so on. But since sometimes it is hard to decide which function a layers fulfills and in order not to make the table too complicated, these layers are grouped under membrane layers.

As for roof structures underneath, in order to separate them from roof structure, they will be named after substrate to emphasize that they lay the foundation and do not necessarily belong to roof. As a result other layers which are already classified as substrate (like the bonding coats in kemper system) will be redirected.

The thermal/ air layer stands for insulation. Since cold roof is not considered, there is no air layer and it will be called simply thermal layer.

	Kemper System Kemper-olV210	Bauder Thermo-plan FPO	FOAMGLAS Compact Roof Systems	NBS Roofs Timber Trussed Rafter	Elastaseal Warm Roof System 20
Finish, Skin	Bonding coats	Polymetric membrane	Wooden grating		Elastaseal top coat
Structure - Joist				Timber trussed rafter	
Structure - Deck			Softwood base boards	Structural veneer plywood	Elastamat glass fibre reinforcement mat
Membrane Layer	Vapour barrier, Bouding coats, Polyurethane waterproof coating	Bitumen sheets	Bitumen sheets	Plastics sheets	Self-adhesive bitumen, Non bitumen based bonding compound
Thermal Layer	PIR foam board	Foam boards	Cellular glass insulation board		Expanded polystyrene board
Substrate			Proprietary concrete	Timber trussed rafter	

Table 2.1: Components warm Roof

Membrane layers, thermal insulation and finish are in most cases modelled, in contrast the load-bearing battens or trussed rafter are not. Partly this is because not all roof types need it. Similarly are bearing layers or deck not always used since bonding coats could partly replace it. Substrates are also not necessary since it belongs more likely to the building. But if the roof is regarded as a whole it should include this part.

In short, the roof layers in BIM from different providers are represented accordingly to reality with details in materials, thickness and sequence with some simplifications. For example the distance and width of joists or rafters are not defined, since they are too detailed and are more related to construction phases. In addition, the mounting materials like screw, bolts are not modelled in the build either. Since the slope of the roof can be variable according to different application, it is certainly not dimensioned in these generic models too. Therefore, for a practical general warm roof structure, it concludes from inside to outside: finish, structure - deck, membrane layer (either for waterproofing or for vapour control), insulation, membrane layer (either for waterproofing or for vapour control) and substrate.

Chapter 3

Analysing BIM-Model with Graph Databases

3.1 Research Question and Approach

For application in the construction industry, one of the most concerning topics is the consistency check of models in BIM, especially the management of level of development and BIM-based model checks. LOD describes the maturity of the model content and lays a foundation for BIM contracts. A precise definition of LOD reduces the risk of miscommunication among members of project teams(BIMForum, 2017). However, project models at any stage of delivery may contain entities at various LODs, causing inconvenience in clarification of the content. Furthermore, BIM models consists of many separately created parts from different disciplines, which may lead to problems during data exchange.

Therefore, in order to obtain a clear picture of the project model, queries about different IFC models on their maturity can be done with Cypher. The consistency of various elements and assemblies at different building phases will be represented and examined with the help of data retrieval queries and algorithms.

To achieve this, the Revit model is first exported into IFC data. To connect the IFC format to the graph database Neo4j, an IFC library that parses internal information and codes to transfer IFC data to Cypher is needed. This is done with the help of the IFC to Cypher code and IfcOpenShell library, this code was first created by user ysangkok in Github and then edited to improve the performance. Then the generated Cypher code is copied into Neo4j and create a graph which represents the Revit model. The following section explains the more detailed workflow will be explained in the following section.

3.2 Used Software, Programs and Libraries

3.2.1 Pycharm

Pycharm is a cross-platform integrated development environment (IDE) mainly used in Python programming. It was developed by the Czech company JetBrains(Wikipedia contributors, 2018b). It provides not only code analysis, code assistant and build tools but also Python debugger. Moreover it supports usage of external libraries, which is important because IFC schema is written in EXPRESS language and it is complicated to extract the stored information in one program.

3.2.2 IfcOpenShell

IfcOpenShell is an open source software library that facilitates users and software developers to work with the IFC file format. This format is commonly used for BIM to describe building and construction data. For geometry information it uses Open Cascade(the Open CASCADE Community Edition) internally to convert the implicit geometry in IFC files into explicit geometry so that other software or modelling packages can understand(IfcOpenShell, 2018). Since the model form is not complicated, geometry parser is not exerted in the above mentioned Python code.

IfcOpenShell now supports IFC2x3 TC1 and IFC4 Add1. The export format used in this thesis is IFC 2x3 coordination view 2.0. IfcOpenShell can be inserted into Autodesk 3ds Max, Blender, BIM server and more importantly, python.

3.2.3 IFC to Neo4j Converter

IFC data can be imported into graph database by putting node properties into an IFC file and entering them in thr Neo4j browser using Cypher language. However, for a large object with numerous properties and accessories, it is time-consuming to do this manually. To automate this process, some IFC to Neo4j converters are available on the internet. Among them is the program, ifc2cypher.py[1], which was concisely written by Github contributor ysangkok in Python language.

The program requires two inputs: second argument (*sys.argv[1]*) as path of IFC file and third argument (*sys.argv[2]*) as label of the nodes. The main idea of program ifc2cypher.py is to use library IfcOpenShell to distribute essential properties and relationships in IFC data. This is done using the *open()* method in IfcOpenShell.

[1]https://gist.github.com/ysangkok/8aa7ab1c3207536518f3c3bf5f664880

```
ourLabel = sys.argv[2]
f = ifcopenshell.open(sys.argv[1])
for el in f:
tid = el.id()
cls = el.is_a()
pairs = []
keys = []
```

Code Listing 3.1: Inputs and main output parameters of ifc2cypher parser

Each object decoded by *ifcopenshell.open* is stored in variant *el*. For those objects in IFC, their associated id numbers and class names are identified separately in *el.tid* and *el.cls*. If the elements have other properties, their keys and values are stored in a dictionary called *pairs*, where values can be indexed by their keys. This step is done using *get_info()* method, which simply means obtaining information, from variant *el*.

After the essential information are organized, cypher commands can be printed with id numbers, class names and other attributes in them.

```
print("CREATE ", end="")
......
print("(a" + str(idx) + ":" + ourLabel + " { nid: " +
str(nId) + ",cls: '" + cls + "'" + pairsStr + " })", end="")
```

Code Listing 3.2: Python Statement to create Nodes in Cypher

A node creating command in cypher, which is printed out by the converter, looks like:

```
CREATE (a1:LOD300{ nid: 1,cls: 'IfcOrganization',
Name: "Autodesk Revit 2017 (ENU)" })
```

Code Listing 3.3: Example of printed Cypher Syntax to create a Node and its Properties in Neo4j

On the other hand, relationships are captured in list *edges*. Nodes and their associate nodes are stored in *destinations*, which uses the *entity_instance* method of IfcOpenShell to detect and analyse connections. The first variant *tid* stands for id of a node and the second *connectedTo* stands for id of its connected node. The last variant *typeDict[cls][i])* indicates relationship type.

```
edges.append((tid, connectedTo, typeDict[cls][i]))
......
for (nId1, nId2, relType) in edges:
print(""" MATCH (a:{:s}),(b:{:s}) WHERE a.nid = {:d}
AND b.nid = {:d}
CREATE (a)-[r:{:s}]->(b) RETURN r; """.format(ourLabel,
ourLabel, nId1, nId2, relType))
```

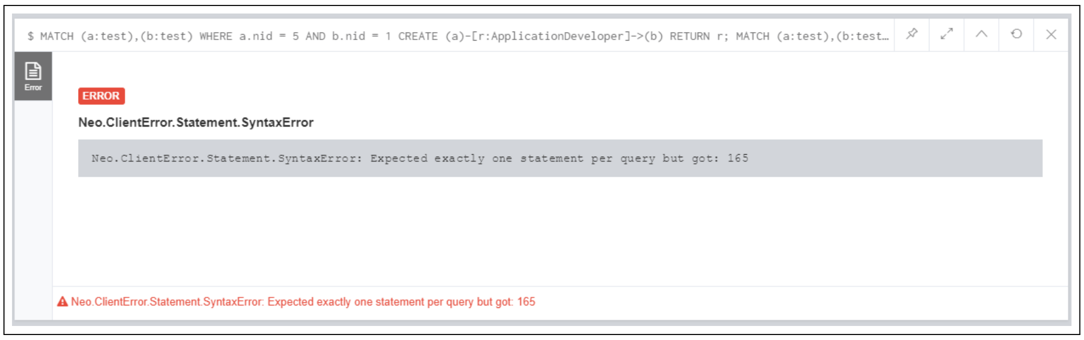
Code Listing 3.4: Python Statement to print Relationships in Cypher

Finally, relationship-creating cypher syntaxes are printed with the information of *edges*. The following is an example of the generated commands:

```
MATCH (a:LOD200),(b:LOD200)
WHERE a.nid = 5 AND b.nid = 1
CREATE (a)-[r:ApplicationDeveloper]->(b)
RETURN r;
......
```

Code Listing 3.5: Example of printed Cypher Syntax to create Relationships between two Nodes in Neo4j

However, because Neo4j only allows one statement consisting of *Match*, *Create* and *Return* command to be executed in one query, and it can be problematic to begin every syntax with *Match* and end it with *Return* while creating relationships. With this statement, all these relationship-creating syntaxes need to be entered and executed individually. This can be a complex and time-consuming work when there are thousands of relationships. Another problem in the original program is that in each relationship-creating process, all the connected nodes are named *a* and *b*, which also leads to the same problem that all syntaxes need to be run separately so as not to mix all relationships in the same nodes. Therefore the way relationship syntaxes are written should be rearranged to improve performance.



```
$ MATCH (a:test),(b:test) WHERE a.nid = 5 AND b.nid = 1 CREATE (a)-[r:ApplicationDeveloper]->(b) RETURN r; MATCH (a:test),(b:test…
```

ERROR

Neo.ClientError.Statement.SyntaxError

```
Neo.ClientError.Statement.SyntaxError: Expected exactly one statement per query but got: 165
```

⚠ Neo.ClientError.Statement.SyntaxError: Expected exactly one statement per query but got: 165

Figure 3.1: Syntax Error

To solve this, *Match* and *Create* needs to be done separately in two lines. Firstly, all the needed nodes are found and matched. the matched nodes should have a unique temporary name. Secondly, relationships are created between them.

```
for (nId1, nId2, relType) in edges:
print("MATCH (a", nId1, ":", ourLabel, "),(b", nId2, ":",
ourLabel, ")
WHERE a", nId1, ".nid = ", nId1, " AND b",nId2, ".nid = ",
nId2, sep='')
for (nId1, nId2, relType) in edges:
print("CREATE (a", nId1, ")-[:", relType, "]->(b", nId2, ")",
sep='')
```

Code Listing 3.6: Improvement of Python Syntax to efficiently create Nodes in Neo4j

The matched nodes are named after the combination of characters and id numbers. This ensures their uniqueness. Furthermore, *Match* syntaxes and *Creat* syntaxes are printed separately to avoid the syntax error that only one statement can exist in one query.

```
MATCH (a5:LOD100),(b1:LOD100) WHERE a5.nid = 5 AND b1.nid = 1
all matches......
CREATE (a5)-[:ApplicationDeveloper]->(b1)
all creates......
```

Code Listing 3.7: Example of improved output Cypher Syntax to create Nodes in Neo4j

3.2.4 Revit 2017

Revit is a software developed by the Autodesk company for BIM. It can be applied in architecture, landscape design, structural engineering, mechanical, electrical and plumbing(MEP) engineering, construction and facility management. It is capable of planing and tracking a building in the whole lifecycle from design to demolition. The main strength of Revit lies in its interoperability with members of an extended project team. Revit can use the IFC, DWG or DGN format to link data from different CAD software. Moreover, it contributes to creating design visualization and document coordination(Khemlani, 2004).

The interoperability is realized by generating smart building data that incorporates not only 3D geometry but also all the relevant data relating to the building and its components(Autodesk, 2018). Autodesk has supported the development of one of the most commonly used smart building data formats - IFC, which lays a good foundation of data exchange not only from Revit into a graph database, but also from many other BIM software programs.

3.2.5 Neo4j Desktop

Neo4j Desktop is a lauchpad for Neo4j applications and tools with automatic software updates. It can connect to production servers and eventually install other components like graph algorithms and graphQL. Local storage of graphs and commands are also possible to ensure convenient data access and analysis. It integrates the Neo4j browser and query manager for querying, visualizing and interacting with graph data using Cypher.

The Neo4j browser is a graphical user interface for writing Cypher and can be used for adding data, running queries, creating relationships, etc. It provides a direct way to visualise the data in the database

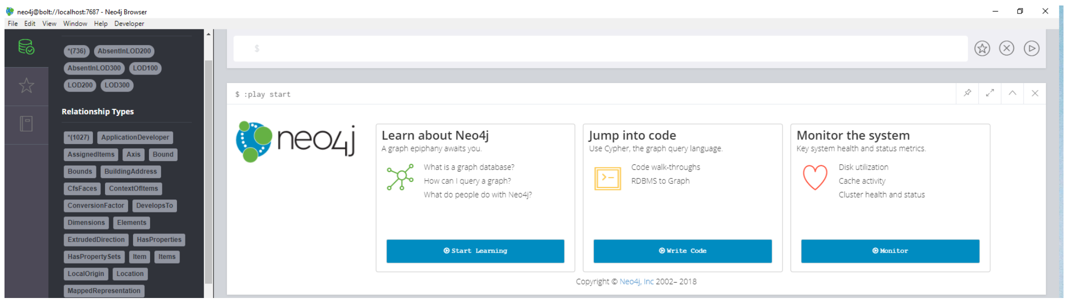

neo4j@bolt://localhost:7687 - Neo4j Browser
File Edit View Window Help Developer
*(736) AbsentInLOD200 AbsentInLOD300 LOD100 LOD200 LOD300
Relationship Types
*(1027) ApplicationDeveloper AssignedItems Axis Bound Bounds BuildingAddress CfsFaces ContextOfItems ConversionFactor DevelopsTo Dimensions Elements ExtrudedDirection HasProperties HasPropertySets Item Items LocalOrigin Location MappedRepresentation
$
$:play start
neo4j
Learn about Neo4j
A graph epiphany awaits you.
What is a graph database?
How can I query a graph?
What do people do with Neo4j?
Start Learning
Jump into code
Use Cypher, the graph query language.
Code walk-throughs
RDBMS to Graph
Write Code
Monitor the system
Key system health and status metrics.
Disk utilization
Cache activity
Cluster health and status
Monitor
Copyright © Neo4j, Inc 2002– 2018

Figure 3.2: Neo4j Browser Interface

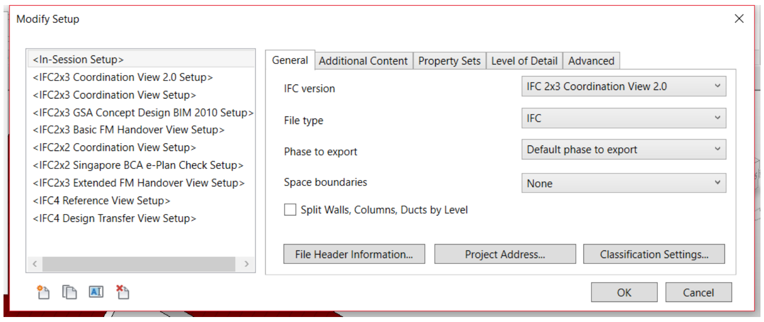
3.2.6 Solibri Model Checker

Solibri Model Checker^{TM} (SMC) is a software tool that analyses BIM for integrity, quality, and physical security. It can x-ray the building model to reveal potential flaws and weaknesses in the design. Furthermore, it is also utilized in highlighting the clashing components and checking that the model complies with the building codes and organizations' own best practices(Solibri, 2014).

SMC enables checking against sets of rules, which is one of its basic concept. Rules can contain building codes, empirical recommendations or individual rule sets that can be customized, such as checking single aspects like sensibility of dimension and checking specific point of view, like usage of correct construction types.

3.3 Importing IFC Data into Graph Databases

First, the Revit model is exported into IFC format. The model view chosen is IFC 2×3 Coordination View Version 2.0. In different export settings different properties sets will be exported. To show the capacity of the graph database generally, the default setting is adopted here.



Modify Setup

<In-Session Setup>
<IFC2x3 Coordination View 2.0 Setup>
<IFC2x3 Coordination View Setup>
<IFC2x3 GSA Concept Design BIM 2010 Setup>
<IFC2x3 Basic FM Handover View Setup>
<IFC2x2 Coordination View Setup>
<IFC2x2 Singapore BCA e-Plan Check Setup>
<IFC2x3 Extended FM Handover View Setup>
<IFC4 Reference View Setup>
<IFC4 Design Transfer View Setup>

General | Additional Content | Property Sets | Level of Detail | Advanced

IFC version	IFC 2x3 Coordination View 2.0
File type	IFC
Phase to export	Default phase to export
Space boundaries	None

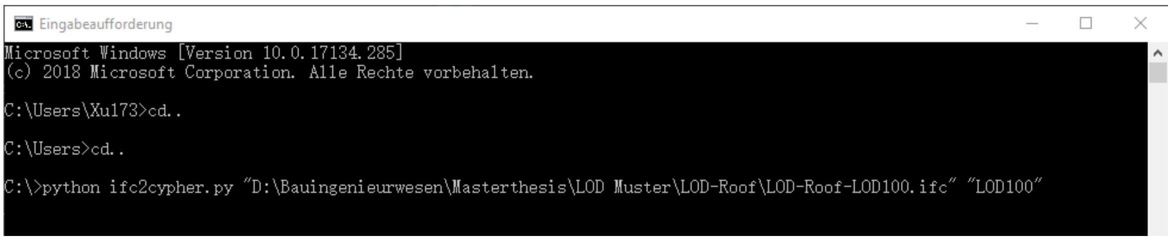
☐ Split Walls, Columns, Ducts by Level

Figure 3.3: Default IFC Export Setup

Second, the user opens the command prompt, goes to the directory of Python command ifc2neo4j.py and uses it to open the exported IFC data. These four arguments in figure 3.4 mean, activating the Python environment, opening the Python program, finding the IFC file in the directory and indicating the labels of the nodes. The labels are entered as their LOD level so that they can work as indicators to when processing nodes with the same class. This example shows the transfer process of the LOD 100 model.

```
python
Test-ifc2cypher.py
"D:\Bauingenieurwesen\Masterthesis\LOD Muster\LOD-Roof\
Test\ModelCheckIntersection.ifc" "ModelCheck"
```

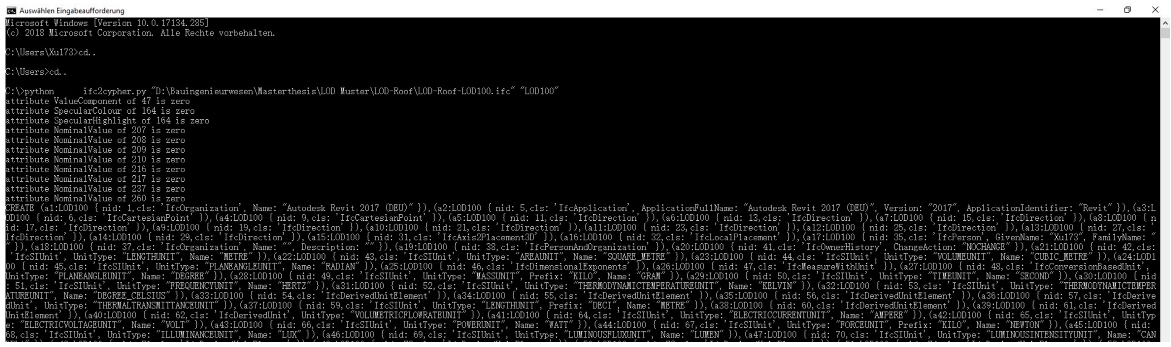
Code Listing 3.8: Arguments given in the Command Prompt



```
Microsoft Windows [Version 10.0.17134.285]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Xu173>cd..

C:\Users>cd..

C:\>python ifc2cypher.py "D:\Bauingenieurwesen\Masterthesis\LOD Muster\LOD-Roof\LOD-Roof-LOD100.ifc" "LOD100"
```

Figure 3.4: Command Prompt Commands

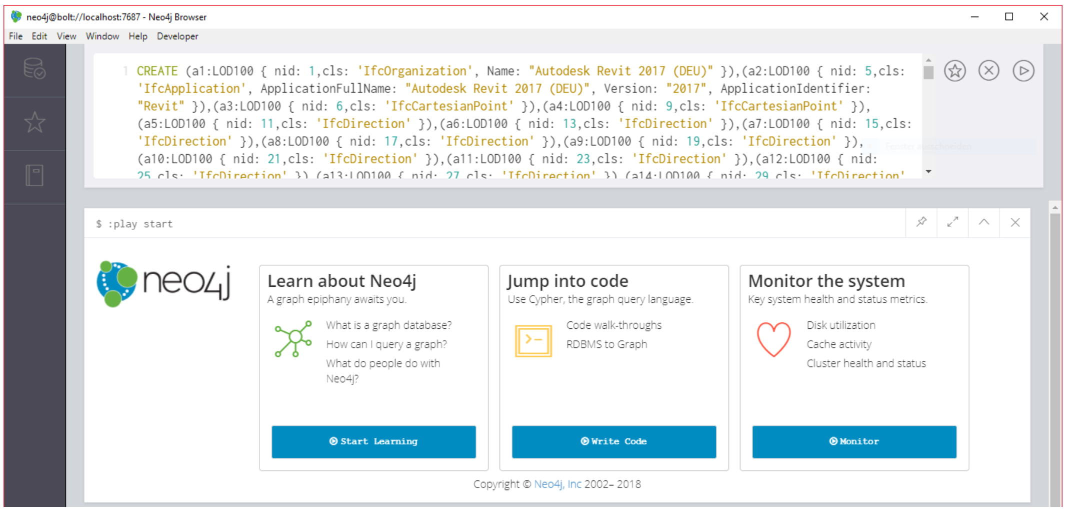
After the command is executed, the Cypher code is generated in the command prompt directly.



```
Microsoft Windows [Version 10.0.17134.285]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Xu173>cd..

C:\Users>cd..

C:\>python     ifc2cypher.py "D:\Bauingenieurwesen\Masterthesis\LOD Muster\LOD-Roof\LOD-Roof-LOD100.ifc" "LOD100"
attribute ValueComponent of 47 is zero
attribute SpecularColour of 164 is zero
attribute SpecularHighlight of 164 is zero
attribute NominalValue of 207 is zero
attribute NominalValue of 208 is zero
attribute NominalValue of 209 is zero
attribute NominalValue of 210 is zero
attribute NominalValue of 216 is zero
attribute NominalValue of 217 is zero
attribute NominalValue of 237 is zero
attribute NominalValue of 260 is zero
CREATE (a1:LOD100 { nid: 1,cls: 'IfcOrganization', Name: "Autodesk Revit 2017 (DEU)" }),(a2:LOD100 { nid: 5,cls: 'IfcApplication', ApplicationFullName: "Autodesk Revit 2017 (DEU)", Version: "2017", ApplicationIdentifier: "Revit" }),(a3:L
OD100 { nid: 6,cls: 'IfcCartesianPoint' }),(a4:LOD100 { nid: 9,cls: 'IfcCartesianPoint' }),(a5:LOD100 { nid: 11,cls: 'IfcDirection' }),(a6:LOD100 { nid: 13,cls: 'IfcDirection' }),(a7:LOD100 { nid: 15,cls: 'IfcDirection' }),(a8:LOD100 { n
id: 17,cls: 'IfcDirection' }),(a9:LOD100 { nid: 19,cls: 'IfcDirection' }),(a10:LOD100 { nid: 21,cls: 'IfcDirection' }),(a11:LOD100 { nid: 23,cls: 'IfcDirection' }),(a12:LOD100 { nid: 25,cls: 'IfcDirection' }),(a13:LOD100 { nid: 27,cls: '
IfcDirection' }),(a14:LOD100 { nid: 29,cls: 'IfcDirection' }),(a15:LOD100 { nid: 31,cls: 'IfcAxis2Placement3D' }),(a16:LOD100 { nid: 32,cls: 'IfcLocalPlacement' }),(a17:LOD100 { nid: 35,cls: 'IfcPerson', GivenName: "Xu173", FamilyName: "
" }),(a18:LOD100 { nid: 37,cls: 'IfcOrganization', Name: "", Description: "" }),(a19:LOD100 { nid: 38,cls: 'IfcPersonAndOrganization' }),(a20:LOD100 { nid: 41,cls: 'IfcOwnerHistory', ChangeAction: "NOCHANGE" }),(a21:LOD100 { nid: 42,cls:
 'IfcSIUnit', UnitType: "LENGTHUNIT", Name: "METRE" }),(a22:LOD100 { nid: 43,cls: 'IfcSIUnit', UnitType: "AREAUNIT", Name: "SQUARE_METRE" }),(a23:LOD100 { nid: 44,cls: 'IfcSIUnit', UnitType: "VOLUMEUNIT", Name: "CUBIC_METRE" }),(a24:LOD1
00 { nid: 45,cls: 'IfcSIUnit', UnitType: "PLANEANGLEUNIT", Name: "RADIAN" }),(a25:LOD100 { nid: 46,cls: 'IfcDimensionalExponents' }),(a26:LOD100 { nid: 47,cls: 'IfcMeasureWithUnit' }),(a27:LOD100 { nid: 48,cls: 'IfcConversionBasedUnit',
UnitType: "PLANEANGLEUNIT", Name: "DEGREE" }),(a28:LOD100 { nid: 49,cls: 'IfcSIUnit', UnitType: "MASSUNIT", Prefix: "KILO", Name: "GRAM" }),(a29:LOD100 { nid: 50,cls: 'IfcSIUnit', UnitType: "TIMEUNIT", Name: "SECOND" }),(a30:LOD100 { nid
: 51,cls: 'IfcSIUnit', UnitType: "FREQUENCYUNIT", Name: "HERTZ" }),(a31:LOD100 { nid: 52,cls: 'IfcSIUnit', UnitType: "THERMODYNAMICTEMPERATUREUNIT", Name: "KELVIN" }),(a32:LOD100 { nid: 53,cls: 'IfcSIUnit', UnitType: "THERMODYNAMICTEMPER
ATUREUNIT", Name: "DEGREE_CELSIUS" }),(a33:LOD100 { nid: 54,cls: 'IfcDerivedUnitElement' }),(a34:LOD100 { nid: 55,cls: 'IfcDerivedUnitElement' }),(a35:LOD100 { nid: 56,cls: 'IfcDerivedUnitElement' }),(a36:LOD100 { nid: 57,cls: 'IfcDerive
dUnit', UnitType: "THERMALTRANSMITTANCEUNIT" }),(a37:LOD100 { nid: 59,cls: 'IfcSIUnit', UnitType: "LENGTHUNIT", Prefix: "DECI", Name: "METRE" }),(a38:LOD100 { nid: 60,cls: 'IfcDerivedUnitElement' }),(a39:LOD100 { nid: 61,cls: 'IfcDerived
UnitElement' }),(a40:LOD100 { nid: 62,cls: 'IfcDerivedUnit', UnitType: "VOLUMETRICFLOWRATEUNIT" }),(a41:LOD100 { nid: 64,cls: 'IfcSIUnit', UnitType: "ELECTRICCURRENTUNIT", Name: "AMPERE" }),(a42:LOD100 { nid: 65,cls: 'IfcSIUnit', UnitTyp
e: "ELECTRICVOLTAGEUNIT", Name: "VOLT" }),(a43:LOD100 { nid: 66,cls: 'IfcSIUnit', UnitType: "POWERUNIT", Name: "WATT" }),(a44:LOD100 { nid: 67,cls: 'IfcSIUnit', UnitType: "FORCEUNIT", Prefix: "KILO", Name: "NEWTON" }),(a45:LOD100 { nid:
68,cls: 'IfcSIUnit', UnitType: "ILLUMINANCEUNIT", Name: "LUX" }),(a46:LOD100 { nid: 69,cls: 'IfcSIUnit', UnitType: "LUMINOUSFLUXUNIT", Name: "LUMEN" }),(a47:LOD100 { nid: 70,cls: 'IfcSIUnit', UnitType: "LUMINOUSINTENSITYUNIT", Name: "CAN
```

Figure 3.5: Generated Cypher Command

The next step is to copy the Cyher command and execute it in the neo4j browser. After the program is successfully run, the complete graph including labels, properties and relationships is generated.


neo4j@bolt://localhost:7687 - Neo4j Browser
File Edit View Window Help Developer
neo4j
Learn about Neo4j
A graph epiphany awaits you.
What is a graph database?
How can I query a graph?
What do people do with Neo4j?
Start Learning
Jump into code
Use Cypher, the graph query language.
Code walk-throughs
RDBMS to Graph
Write Code
Monitor the system
Key system health and status metrics.
Disk utilization
Cache activity
Cluster health and status
Monitor
Copyright © Neo4j, Inc 2002– 2018

```
CREATE (a1:LOD100 { nid: 1,cls: 'IfcOrganization', Name: "Autodesk Revit 2017 (DEU)" }),(a2:LOD100 { nid: 5,cls:
'IfcApplication', ApplicationFullName: "Autodesk Revit 2017 (DEU)", Version: "2017", ApplicationIdentifier:
"Revit" }),(a3:LOD100 { nid: 6,cls: 'IfcCartesianPoint' }),(a4:LOD100 { nid: 9,cls: 'IfcCartesianPoint' }),
(a5:LOD100 { nid: 11,cls: 'IfcDirection' }),(a6:LOD100 { nid: 13,cls: 'IfcDirection' }),(a7:LOD100 { nid: 15,cls:
'IfcDirection' }),(a8:LOD100 { nid: 17,cls: 'IfcDirection' }),(a9:LOD100 { nid: 19,cls: 'IfcDirection' }),
(a10:LOD100 { nid: 21,cls: 'IfcDirection' }),(a11:LOD100 { nid: 23,cls: 'IfcDirection' }),(a12:LOD100 { nid:
25,cls: 'IfcDirection' }),(a13:LOD100 { nid: 27,cls: 'IfcDirection' }),(a14:LOD100 { nid: 29,cls: 'IfcDirection'
```

```
$ :play start
```

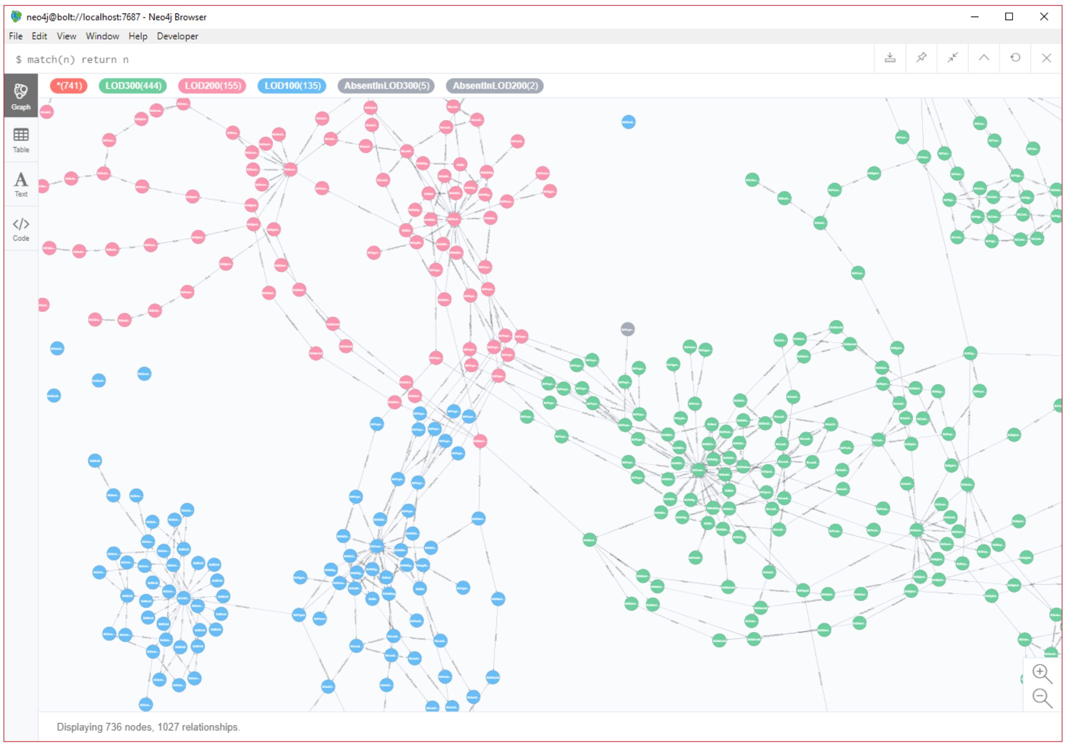
Figure 3.6: Entering Cypher Command in Neo4j Browser

To match and compare three LOD models to each other, the graph of all three LODS need to be created and strored in the same database. Using one graph database for multiple models enables not only a comparison study but also history traces. Therefore this procedure is repeated for the LOD 200 and LOD 300 models in the same browser. Finally a complete graph with all the essential information from all three LODs is created. The generation of model check object uses the same mechanism too. To examine the dataset the following command can be used.

```
MATCH(n) RETURN n
```

Code Listing 3.9: Cypher Syntax to verify transferred IFC relationship

The LOD 100 node are displayed in blue and the LOD 200 nodes are in red and the LOD 300 nodes are in green. The names on the nodes stand for the class names, which make it easier to distinguish their functions. Properties of the associated classes are stored in the node and can be easily viewed by clicking on the nodes. The lines between nodes represent the relationships between the nodes and their names indicate the specific type. The crossed lines do not lead to crossed relationships since they are all one-to-one.



```
$ match(n) return n
```

Displaying 736 nodes, 1027 relationships.

Figure 3.7: Graph of three LODs

Chapter 4

Property Queries and its Application in Graph Database

In this chapter, the potential of using graph databases to visualize and analyse BIM models and their level of developments will be demonstrated by several examples. The queries are mainly organized in increasing terms of complexity into three sections. The first section introduces simple queries on checking basic attributes and properties of the LOD-models. These properties include geometry, openings and layers of the roof. This step is performed to prove that graph databases has the capability in solving basic information retrieving tasks which other traditional databases and BIM software can do. The next step is to add some properties that are lost due to basic export settings, like essential node coordinates and detailed thermal mass data. These added properties are selected so that the capacity of the property query method, which will be discussed in the third section, can be fully demonstrated. The following section investigates the potential of graph databases in comparing and analysing semantic information in various development stages, so as to present the advantage of graphs in visualizing evolving process.

4.1 Anaylsis of Properties

To lay the foundation of advanced queries, it is essential to evaluate the capability of graph databases in retrieving basic geometric and semantic information of a BIM model. Moreover, because all the nodes are labelled with their LODs, the results can be listed according to their degree of maturity, so as to completely demonstrate the evolving process of building models.

4.1.1 Retrieve the Length and Width of the Roof

The geometry of the roof is stored in class *IfcRectangleProfileDef*. By searching for the nodes with *IfcRectangleProfileDef* class and extracting their information, the length and width can be found. However, size of openings like windows are also defined by same class, which will lead to misunderstandings. In order to exclude openings when querying roofs, differences between these two types should be found.

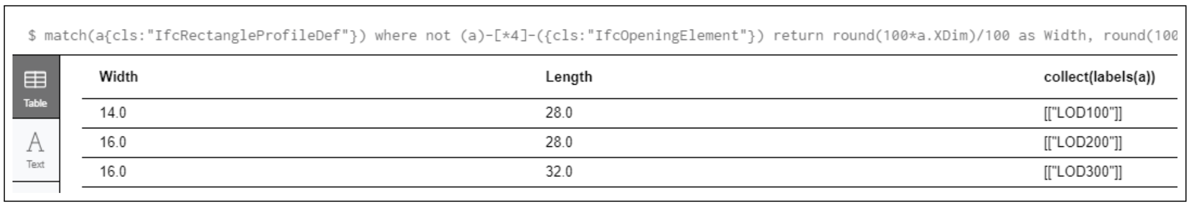
```
#147= IFCRECTANGLEPROFILEDEF(.AREA.,'Roof LOD 300',#146,
15.9999999999999, 32.0000000000001);
```

Code Listing 4.1: Example IFC expression of Roof Profile

```
#183= IFCRECTANGLEPROFILEDEF(.AREA.,'Roof LOD 300',#182,1.,1.5)
```

Code Listing 4.2: Example IFC expression of Opening Profile

Traditional relational databases rely on organizing primary keys of different tables in a key table to manage many to many relationships, which is sometimes inefficient in Ifc files since Ifc data is organized hierarchically and contain a numerous relationships. Additionally, relational database need to search between columns, rows of tables to find the queried data. In contrast, graph database can take advantage of directional relationships and pattern filter to quickly narrow research. For instance, opening profile definition have a specific 4-degree relationship with *IfcOpeningElement* (*IfcRectangleProfileDef* – *IfcExtrudedAreaSolid* – *IfcShapeRepresentation* – *IfcProductDefinitionShape* - *IfcOpeningElement*). In Neo4j, this feature can be used as a filter. If all the profile classes with such pattern are excluded, then the remaining definitions are for roof elements.


```
$ match(a{cls:"IfcRectangleProfileDef"}) where not (a)-[*4]-({cls:"IfcOpeningElement"}) return round(100*a.XDim)/100 as Width, round(100
```

Width	Length	collect(labels(a))
14.0	28.0	[["LOD100"]]
16.0	28.0	[["LOD200"]]
16.0	32.0	[["LOD300"]]

Figure 4.1: Result Length and Width Query

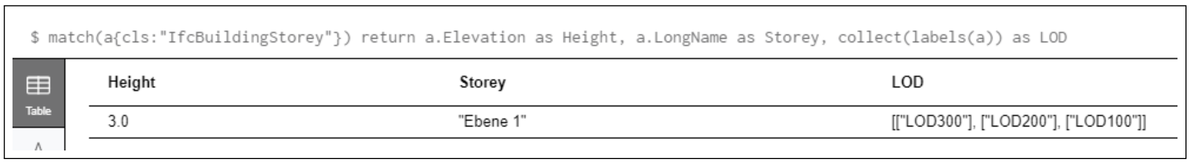
```
MATCH(a{cls:"IfcRectangleProfileDef"})
WHERE NOT (a)-[*4]-({cls:"IfcOpeningElement"})
RETURN ROUND(100*a.XDim)/100 AS Width,
ROUND(100*a.YDim)/100 AS Length,
COLLECT(LABELS(a))
```

Code Listing 4.3: Match and return the rounded roof Dimensions while filtering out openings

From the table the size of roof during evolving process are shown clearly (from 14m×28m in LOD 100 to 16m×32m in LOD 300). The figures are rounded to avoid numerous digits owing to high accuracy.

4.1.2 Retrieve Height of the Roof

Height and the Storeys of the roofs are in are defined in class *IfcBuildingStorey*, one under property *Elevation* and the other under property *LongName*. Because all the roofs have the same height and level, the results are displayed in one line.



```
$ match(a{cls:"IfcBuildingStorey"}) return a.Elevation as Height, a.LongName as Storey, collect(labels(a)) as LOD
```

Height	Storey	LOD
3.0	"Ebene 1"	[["LOD300"], ["LOD200"], ["LOD100"]]

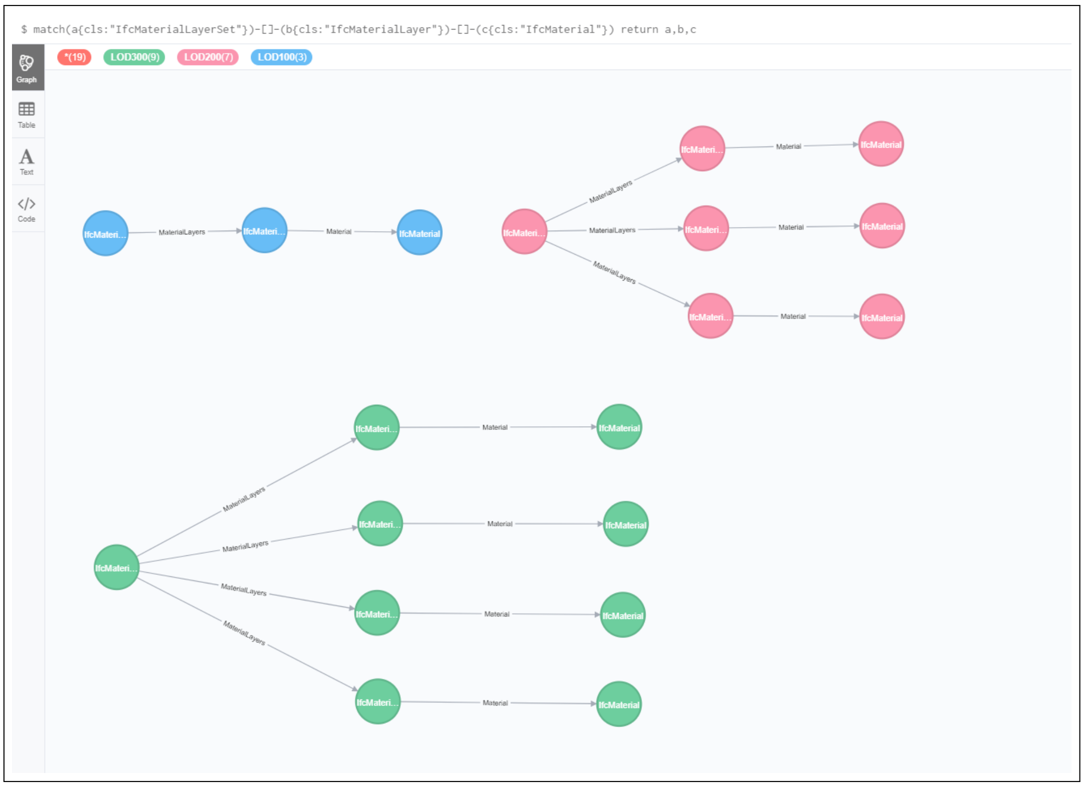
Figure 4.2: Result Height and Storey Query

```
MATCH(a{cls:"IfcBuildingStorey"})
RETURN a.Elevation AS Height, a.LongName AS Storey,
COLLECT(LABELS(a)) AS LOD
```

Code Listing 4.4: Match and return the roof Elevation and storey Name

4.1.3 Retrieve Layers and Thickness of the Roof

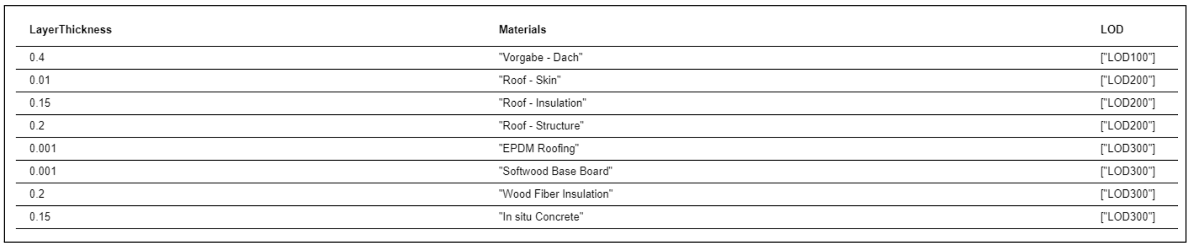
Class that contain roof layer thickness are called *IfcMaterialLayer*. On the other hand, *IfcMaterial* defines its Material. Because each layer thickness and material information are defined separately, a direct retrieval of these two classes may cause misunderstanding. However, in graph databases *IfcMaterialLayer* and its associate *IfcMaterial* are linked together. By searching for pattern *IfcMaterialLayer-IfcMaterial* instead of nodes can clearly organize the result and avoid chaos. This is more convenient than looking for primary keys of tables in relational databases to sort out wrong information. Following is illustration of relationships between thickness data and material data in three LODs, which is led by *IfcMaterialLayerSet* of each LOD.



```
$ match(a{cls:"IfcMaterialLayerSet"})-[]-(b{cls:"IfcMaterialLayer"})-[]-(c{cls:"IfcMaterial"}) return a,b,c
```

Figure 4.3: Relationships of Layer Thickness and Material

It demonstrates the capability of graph databases that not only can it filter information by keywords, but more importantly, by patterns. This is realized by utilizing object-oriented data and objectified relationship in IFC data, as well as advantages of graph databases of processing relationships.



LayerThickness	Materials	LOD
0.4	"Vorgabe - Dach"	["LOD100"]
0.01	"Roof - Skin"	["LOD200"]
0.15	"Roof - Insulation"	["LOD200"]
0.2	"Roof - Structure"	["LOD200"]
0.001	"EPDM Roofing"	["LOD300"]
0.001	"Softwood Base Board"	["LOD300"]
0.2	"Wood Fiber Insulation"	["LOD300"]
0.15	"In situ Concrete"	["LOD300"]

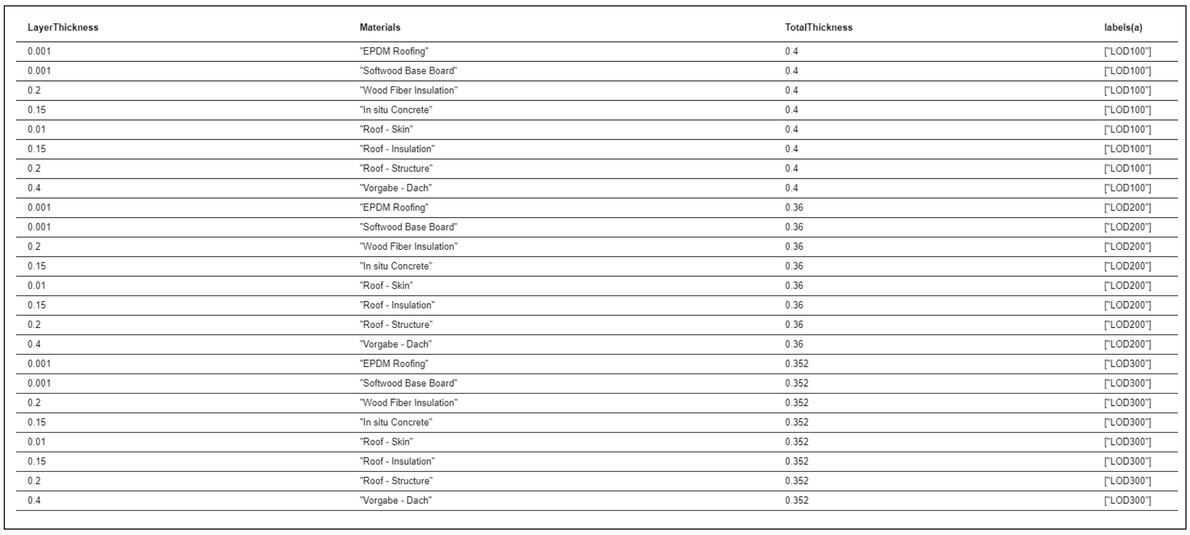
Figure 4.4: Result Layer Material and Thickness Query

```
MATCH(a{cls:"IfcMaterialLayer"})-[]-(b{cls:"IfcMaterial"})
RETURN a.LayerThickness AS LayerThickness,
b.Name AS Materials,
LABELS(a) AS LOD ORDER BY LABELS(a)
```

Code Listing 4.5: Match and return the Material and its Thickness of Roofs in different LODs

The materials and their associate thickness are listed from low LOD to high LOD. This demonstrates the evolution of roof materials directly. In the beginning, roof material is not assigned and thus *Vorgabe - Dach* is shown. But the rough thickness (0.4m) is indicated. In LOD 200, essential layers are defined as skin, insulation and structure, but the specific materials are still unknown. Finally in the LOD 300 stage, specific materials are named including their detailed thickness. The increase of decimal digits also indicates higher accuracy and maturity.

This can be combined by total thickness query. Unlike layer thickness, total thickness is stored in *IfcExtrudedAreaSolid*. Therefore it needs to be matched separately too.



LayerThickness	Materials	TotalThickness	labels(a)
0.001	"EPDM Roofing"	0.4	["LOD100"]
0.001	"Softwood Base Board"	0.4	["LOD100"]
0.2	"Wood Fiber Insulation"	0.4	["LOD100"]
0.15	"In situ Concrete"	0.4	["LOD100"]
0.01	"Roof - Skin"	0.4	["LOD100"]
0.15	"Roof - Insulation"	0.4	["LOD100"]
0.2	"Roof - Structure"	0.4	["LOD100"]
0.4	"Vorgabe - Dach"	0.4	["LOD100"]
0.001	"EPDM Roofing"	0.36	["LOD200"]
0.001	"Softwood Base Board"	0.36	["LOD200"]
0.2	"Wood Fiber Insulation"	0.36	["LOD200"]
0.15	"In situ Concrete"	0.36	["LOD200"]
0.01	"Roof - Skin"	0.36	["LOD200"]
0.15	"Roof - Insulation"	0.36	["LOD200"]
0.2	"Roof - Structure"	0.36	["LOD200"]
0.4	"Vorgabe - Dach"	0.36	["LOD200"]
0.001	"EPDM Roofing"	0.352	["LOD300"]
0.001	"Softwood Base Board"	0.352	["LOD300"]
0.2	"Wood Fiber Insulation"	0.352	["LOD300"]
0.15	"In situ Concrete"	0.352	["LOD300"]
0.01	"Roof - Skin"	0.352	["LOD300"]
0.15	"Roof - Insulation"	0.352	["LOD300"]
0.2	"Roof - Structure"	0.352	["LOD300"]
0.4	"Vorgabe - Dach"	0.352	["LOD300"]

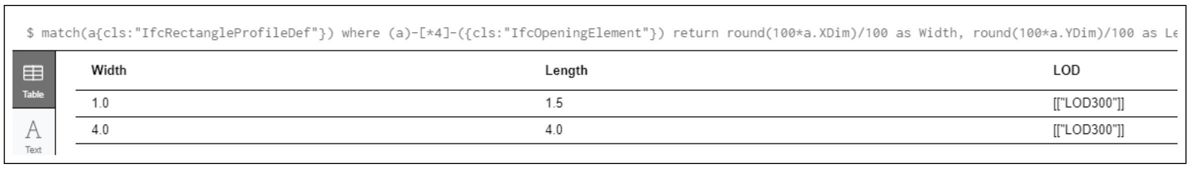
Figure 4.5: Result Layer and total Thickness Query

```
MATCH(c{cls:"IfcMaterialLayer"})-[]-(d{cls:"IfcMaterial"}),
(a{cls:"IfcRectangleProfileDef"}),
(a)-[]-(b{cls:"IfcExtrudedAreaSolid"})
WHERE NOT (a)-[*4]-({cls:"IfcOpeningElement"})
RETURN c.LayerThickness AS LayerThickness,
d.Name AS Materials,
b.Depth AS TotalThickness,
LABELS(a) ORDER BY LABELS(a)
```

Code Listing 4.6: Match and return the Material, its Thickness and total Thickness of Roofs in different LODs

4.1.4 Retrieve Geometry and Number of Opeinings

From the first section the method of separating roofs from openings is introduced to display only the geometry of the roofs. Inversely geometry of openings exclusively can be listed by changing the filter from *where not* to *where*.



```
$ match(a{cls:"IfcRectangleProfileDef"}) where (a)-[*4]-({cls:"IfcOpeningElement"}) return round(100*a.XDim)/100 as Width, round(100*a.YDim)/100 as Le
```

Width	Length	LOD
1.0	1.5	[["LOD300"]]
4.0	4.0	[["LOD300"]]

Figure 4.6: Result Openings Geometry Query

```
MATCH(a{cls:"IfcRectangleProfileDef"})
WHERE (a)-[*4]-({cls:"IfcOpeningElement"})
RETURN ROUND(100*a.XDim)/100 AS Width,
ROUND(100*a.YDim)/100 AS Length,
COLLECT(LABELS(a)) AS LOD
```

Code Listing 4.7: Match and return the rounded Opening Dimensions

In this example, there are two types of openings in LOD 300 - shafts and windows. Unlike windows, shafts are just a simple opening and have no framing, lining or glazing. In IFC, all openings, including windows and shafts, are defined by *IfcOpeningElement*.

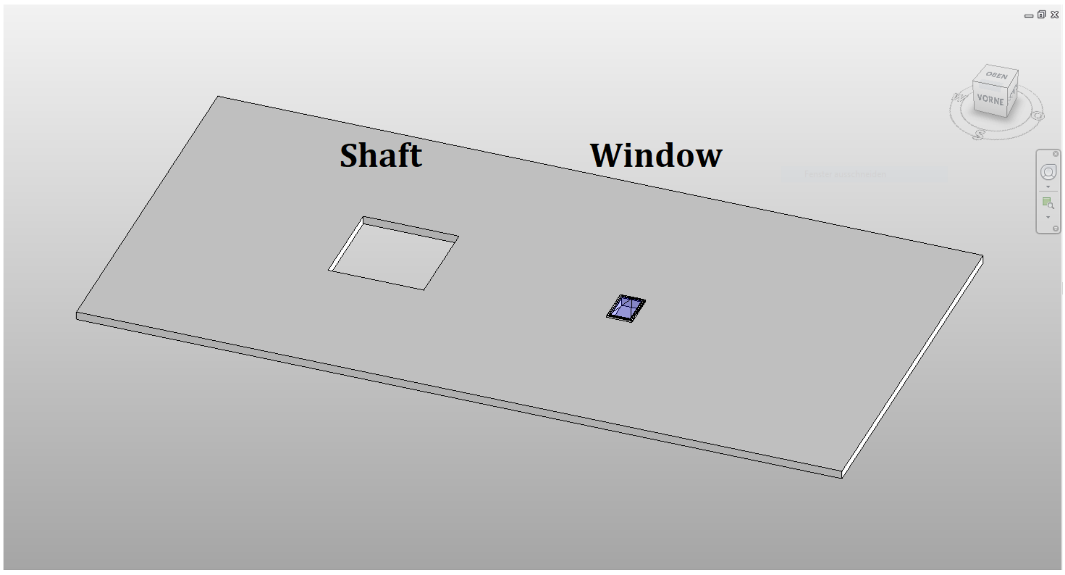
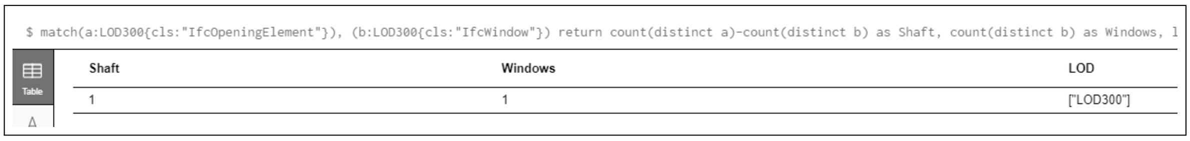


Figure 4.7: Difference Shaft and Window

A complex structure may include numerous windows and openings. It is also useful to count the number of them. This can be done in a simpler way using the assumption that number of openings = number of shaft + number of windows. Firstly number of all *IfcOpeningElements* and *IfcWindows* will be detected, then minus number of openings by number of windows to get number of shafts. Eventually can other kinds of openings like doors also be considered with the help of *Ifc*-elements.



```
$ match(a:LOD300{cls:"IfcOpeningElement"}), (b:LOD300{cls:"IfcWindow"}) return count(distinct a)-count(distinct b) as Shaft, count(distinct b) as Windows, l
```

Shaft	Windows	LOD
1	1	["LOD300"]

Figure 4.8: Result Number of Openings Query

```
MATCH
(a:LOD300{cls:"IfcOpeningElement"}),(b:LOD300{cls:"IfcWindow"})
RETURN COUNT(distinct a)-COUNT(distinct b) AS Shaft,
COUNT(distinct b) AS Windows, labels(a) AS LOD
```

Code Listing 4.8: Match and return the Number of different Opening Types

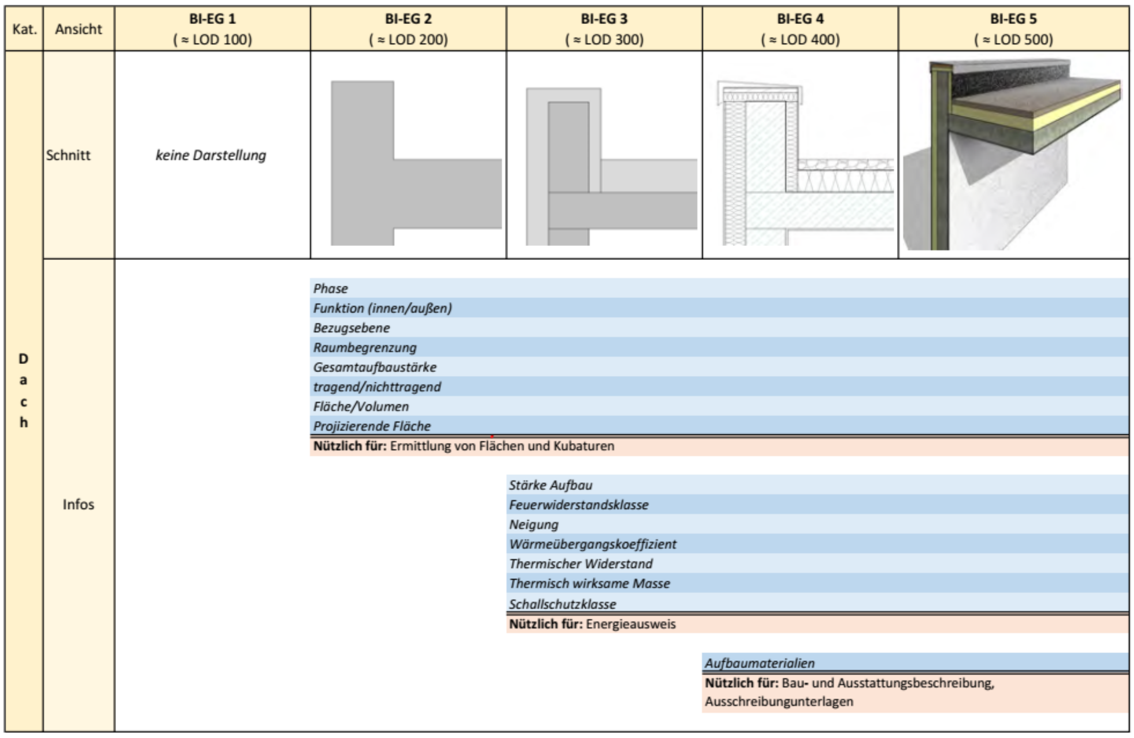
4.2 Reliability Check of the LOD-Model

LOD defines the essential information content and its degree of clarity. Information content can contain two types of information: a) the element's geometry and b) associated numeric and/or textual attributes(BIMForum, 2017). Besides general geometry properties, an element can contain many other specific information points. For example a door might contain information about whether it is smoke-stop, whether it is self-closing, its opening direction and its manufacturer. These properties are required particularly according to different demands of projects or design individuals. Therefore it takes more time and effort to track and examine whether the right property set is defined in the right design stage. An absent attribute may cause trouble when it is needed while an early defined attribute may be unreliable. To identify these possible problems, the capability of graph database can be tested.

4.2.1 Instance of Roof Information Specification in Different LOD

First, to fully demonstrate the practicability of a property set query using graph databases in more pragmatic conditions, a realistic property requirement standard needs to be set. Some institutions have created matrices and tables to manage BIM data to specifically define what objects and properties should be in a model that is modelled on a certain LOD. However, few classification systems have specific definitions when it comes to detailed descriptions of elements like roofs, doors and walls. One example is the "BI-EG" standard (Build Informed-

Entwicklungsgrade which is in accordance with the LOD specification by BIMForum) created by Austrian BIM company BIM Informed GmbH.



Kat.	Ansicht	BI-EG 1 (≈ LOD 100)	BI-EG 2 (≈ LOD 200)	BI-EG 3 (≈ LOD 300)	BI-EG 4 (≈ LOD 400)	BI-EG 5 (≈ LOD 500)
Dach	Schnitt	*keine Darstellung*				
	Infos		*Phase*			
			Funktion (innen/außen)			
			Bezugsebene			
			Raumbegrenzung			
			Gesamtaufbaustärke			
			tragend/nichttragend			
			Fläche/Volumen			
			Projizierende Fläche			
			Nützlich für: Ermittlung von Flächen und Kubaturen			
				Stärke Aufbau		
				Feuerwiderstandsklasse		
				Neigung		
				Wärmeübergangskoeffizient		
				Thermischer Widerstand		
				Thermisch wirksame Masse		
				Schallschutzklasse		
				Nützlich für: Energieausweis		
					Aufbaumaterialien	
					Nützlich für: Bau- und Ausstattungsbeschreibung, Ausschreibungunterlagen	

Figure 4.9: Property Specification of Roof in different LOD by Bim Informed GmbH(Hörtnagl, 2017)

According to interpretation of BIMFroum, LOD 100 elements are not geometric representations and all information must be considered approximate. Therefore there are no requirements in LOD 100. Meanwhile because major characteristics of components in LOD 200 models allow quick takeoffs, like their thickness and width, theses quantities should be obtained(Bloomberg *et al.*, 2012). Therefore geometric and basic textual information can be retrieved in LOD 200. A LOD 300 model can be used for analysis such as: energy performance, clash and cost and specific takeoffs. This requires more well-defined characteristics about dfferent components(Bloomberg *et al.*, 2012). Thus, more attached specifications are required in LOD 300, like thermal performance and fire rating.

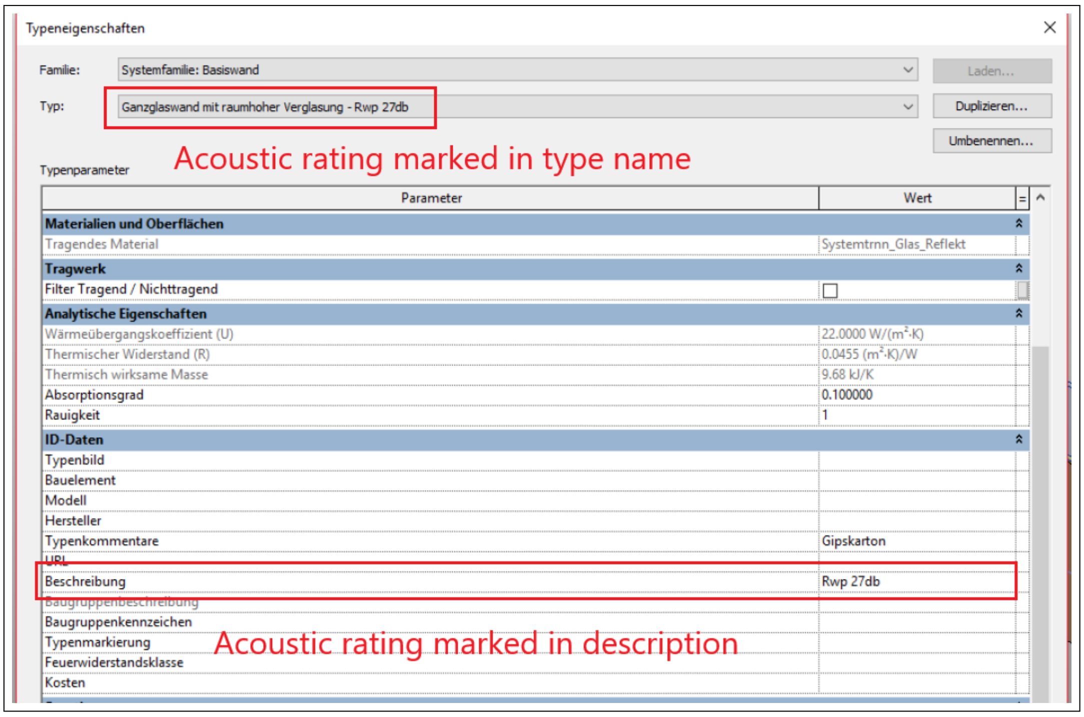
According to BIMForum, an LOD 400 element is modeled in sufficient detail and sufficient for fabrication, assembly and installation, which are generally not included in digital models in common BIM projects. Finally, LOD 500 is a field verified representation and it is not defined in the specification or in the BIM Informed requirements. Therefore the property set check of BIM models is conducted within LOD 100, 200 and 300.

LODs	BIMForum fundamental LOD Definitions	Bim Informed Roof Information Specification
LOD100	The Model Element may be graphically represented in the Model with a symbol or other generic representation, but does not satisfy the requirements for LOD 200. Information related to the Model Element (i.e. cost per square foot, tonnage of HVAC, etc.) can be derived from other Model Elements.	none
LOD200	The Model Element is graphically represented within the Model as a generic system, object, or assembly with approximate quantities, size, shape, location, and orientation. Non-graphic information may also be attached to the Model Element.	phase, external/internal, number of storey, room boundaries/not room boundaries, total thickness, load bearing/not load bearing, area/volume, projected area
LOD300	The Model Element is graphically represented within the Model as a specific system, object or assembly in terms of quantity, size, shape, location, and orientation. Non-graphic information may also be attached to the Model Element.	material layer thickness, fire rating, pitched angle, thermal transmittance, thermal resistance, thermal mass, acoustic rating

Table 4.1: Comparison between BIMForum LOD Specififation and Bim Informed Roof Definition

4.2.2 Definition of Property Set in IFC

Different design individuals have different protocols to point out those specific numeric/textual attributes. In general, these attributes are entered in properties in the form of comments, descriptions or parameters. However, some mark them in the element name instead. Here it is assumed that attributes like thermal transmittance are stored in the element property.



Typeneigenschaften

Familie: Systemfamilie: Basiswand

Typ: Ganzglaswand mit raumhoher Verglasung - Rwp 27db

Laden...

Duplizieren...

Umbenennen...

Typenparameter

Acoustic rating marked in type name

Parameter	Wert
Materialien und Oberflächen	
Tragendes Material	Systemtrnn_Glas_Reflekt
Tragwerk	
Filter Tragend / Nichttragend	☐
Analytische Eigenschaften	
Wärmeübergangskoeffizient (U)	22.0000 W/(m²·K)
Thermischer Widerstand (R)	0.0455 (m²·K)/W
Thermisch wirksame Masse	9.68 kJ/K
Absorptionsgrad	0.100000
Rauigkeit	1
ID-Daten	
Typenbild	
Bauelement	
Modell	
Hersteller	
Typenkommentare	Gipskarton
URL	
Beschreibung	Rwp 27db
Baugruppenbeschreibung	
Baugruppenkennzeichen	
Typenmarkierung	
Feuerwiderstandsklasse	
Kosten	

Acoustic rating marked in description

Figure 4.10: Example Attribute in Type Name and Description

The top part of Figure 4.10 is an example of storing sound insulation information in the name of the object, the bottom is a demonstration of marking it in the description row. Both methods are widely adopted in the BIM industry.

To query these attributes, it is firstly essential to know how they are defined. In IFC data, specific properties like thermal performance and, fire rating are described mainly by IfcPropertySingleValue, which is aggregated within a property tree in the container class *IfcPropertySets*. With the help of the objectified relationship *IfcRelDefinesByProperties* property sets are assigned to relevant objects such as slabs, roofs or windows. Elements can be represented by a set of predefined property definitions. According to buildingSMART definition, property sets can be either directly assigned to occurrence objects using this relationship or assigned to an object type and, via that type, to occurrence objects. It can also be assigned to multiple objects using the 1-to-N *IfcRelDefinesByProperties*.

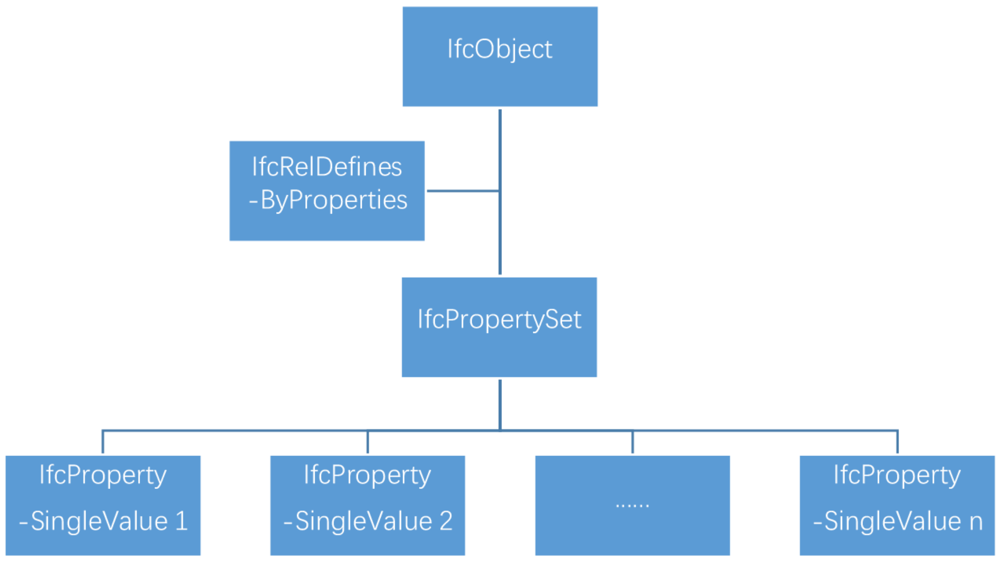
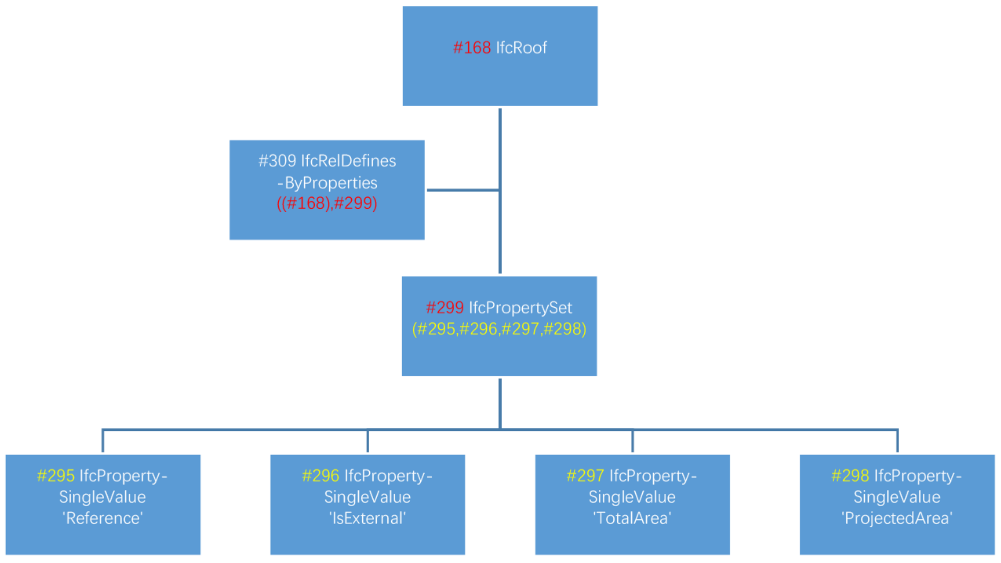

IfcObject
IfcRelDefines -ByProperties
IfcPropertySet
IfcProperty -SingleValue 1
IfcProperty -SingleValue 2
......
IfcProperty -SingleValue n

Figure 4.11: Relationship between Properties and Object in IFC


#168 IfcRoof
#309 IfcRelDefines -ByProperties ((#168),#299)
#299 IfcPropertySet (#295,#296,#297,#298)
#295 IfcProperty-SingleValue 'Reference'
#296 IfcProperty-SingleValue 'IsExternal'
#297 IfcProperty-SingleValue 'TotalArea'
#298 IfcProperty-SingleValue 'ProjectedArea'

The node number in *IfcRelDefinesByProperties* stands for the particular element and its relevant property set. The definition of *IfcPropertySet* contains its associate single values.

Figure 4.12: Example of Relationship between Properties and Object in IFC

One exception is the thickness and material. Both total thickness and layer thickness and material are contained in *IfcMaterialLayerSet*. It shares the same data structure as property set. *IfcMaterialLayer* defines the material and its thickness. Therefore if *IfcMaterialLayerSet* is detected, relevant thickness information is indicated.

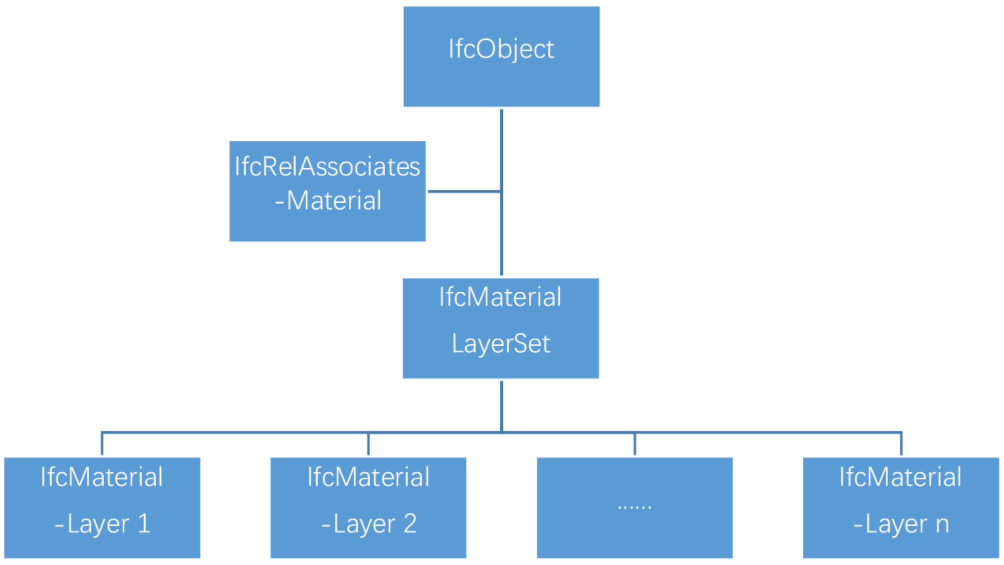
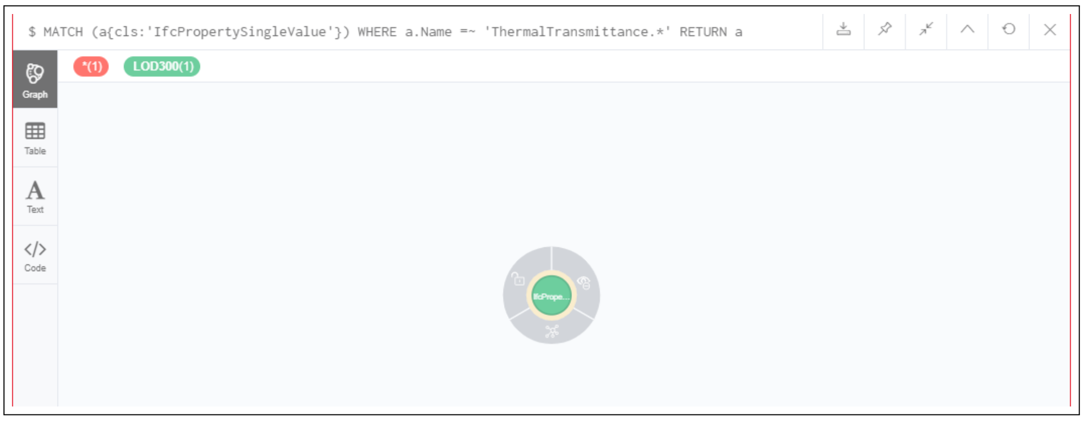


Figure 4.13: Relationship between Material Layers and Object in IFC

4.2.3 Retrieve Property Set

By matching single property values and filtering them by names desired property can be retrieved. For example, if the thermal transmittance of the roof slab must be checked, every node containing "ThermalTransmittance" in its name is retrieved.



```
$ MATCH (a{cls:'IfcPropertySingleValue'}) WHERE a.Name =~ 'ThermalTransmittance.*' RETURN a
```

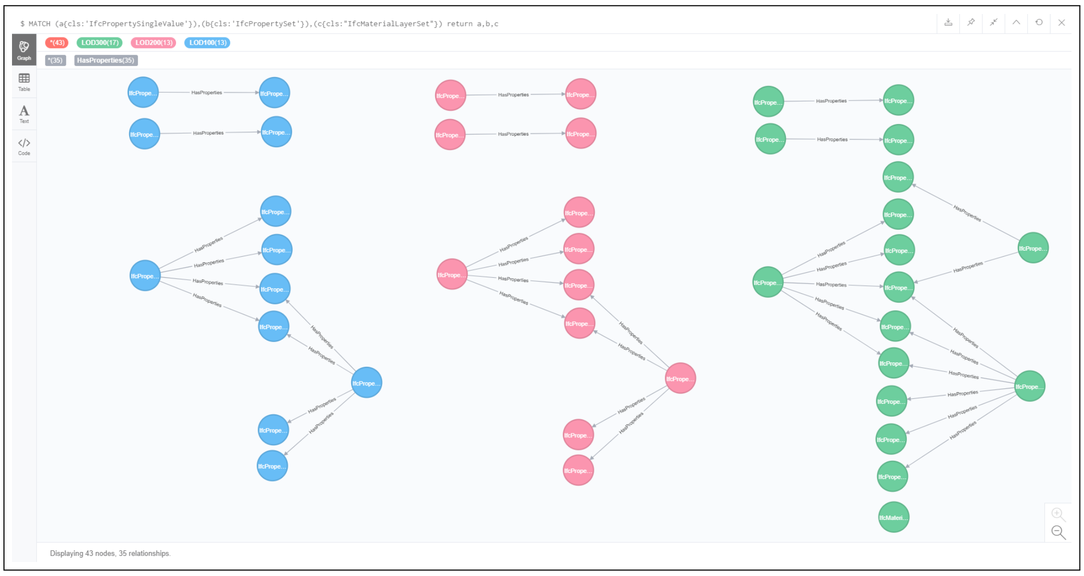
Figure 4.14: Result Thermal Transmittance Query

```
MATCH (a{cls:'IfcPropertySingleValue'})
WHERE a.Name =~ 'ThermalTransmittance.*'
RETURN a
```

Code Listing 4.9: Match and return Properties which are relevant to thermal Transmittance

The one green node indicates that thermal transmittance is defined only in LOD 300. There is no definition of thermal mass and thermal resistance, therefore, only one node is called ThermalTransmittance.

To determine all the numeric and textural properties and the thickness information, all the property single values and material layer sets can be queried. Moreover, *IfcPropertySets* are also needed to indicate what these values belong to.



```
$ MATCH (a{cls:'IfcPropertySingleValue'}),(b{cls:'IfcPropertySet'}),(c{cls:"IfcMaterialLayerSet"}) return a,b,c
```

Displaying 43 nodes, 35 relationships.

Figure 4.15: Result Property Set, Property single Value and Material Layer Set Query

```
MATCH
(a{cls:'IfcPropertySingleValue'}),(b{cls:'IfcPropertySet'}),
(c{cls:"IfcMaterialLayerSet"})
RETURN a,b,c
```

Code Listing 4.10: Match and return Property Sets and Material Layer Sets

The nodes at the left and right of each LOD stand for property sets. In LOD 100 and 200, there are property sets for slab, roof, building and building storey. The window set is first created in LOD 300. Because there is only one material layer set in each LOD, it can be easily queried without distinguishing which object it belongs to. In a building with many elements there might be many kinds of materials from different elements, this can also be easily filtered after the layer set name.

The graph displays all the existing attributes. The evolution and development of property nodes can be easily observed. The LOD 100 and 200 model contains the same amount of properties since once the model is created, basic and essential items like geometry are automatically generated. The number of nodes increase in LOD 300 because of newly introduced attributes and window elements. However, these nodes do not correspond to required properties and need to be matched using roof information specification.

4.2.4 Match Property Sets

Compared to evaluating a model and assigning a level to it, it makes more sense to check whether the minimum required information is achieved in a model in an LOD level. There needs to be a minimum amount of information in a model to be able to use it for a task, more exactly, the real message modellers that give to team members is about the level of reliability of the data(Van Berlo & Bomhof, 2014). The goal of analysing property sets is to develop a method or protocol that evaluates the information that should be and does not need to be in a model to perform a specific task. On the other hand to fulfil the function of collaboration facilitation between various project participants, it is essential to know the level of reliability of the model before it is delivered to the next stage or to the next team. Furthermore, it is also important to distinguish in which stage these properties are needed.

The next step is to distinguish the classified aim of the query, what properties are missing, which LOD they missing since, and what properties are defined in advance. The following types of properties concern presence sequence and reliability:

Property Type	Subtype	Reliability
Set in the required LOD	required in LOD 100/200/300	reliable
Set in advance before the required LOD	required in LOD 200, defined in LOD100/ required in LOD 300, defined in LOD 200/ required in LOD 300, defined in LOD 100	unreliable, can be taken as approximate or predefined specification
Absent from the required LOD but appear later	Missing in LOD 200, defined in LOD 300/ Missing in LOD 100, defined in LOD 200/ Missing in LOD 100, defined in LOD 300	undeliverable, required information missing, but is reliable once made up
Absent since required LOD until the end	Missing since LOD 100/200/300	undeliverable, required information missing

Table 4.2: Property type based on reliability and sequence

Since there are various property types and levels of reliability, it is complex and difficult to understand how to group and classify attributes with rows and columns in tables, as relational dabases do. Graph databases offers an explicit and clear way to illustrate and distinguish them by patterns and colours.

The whole workflow is as follows:

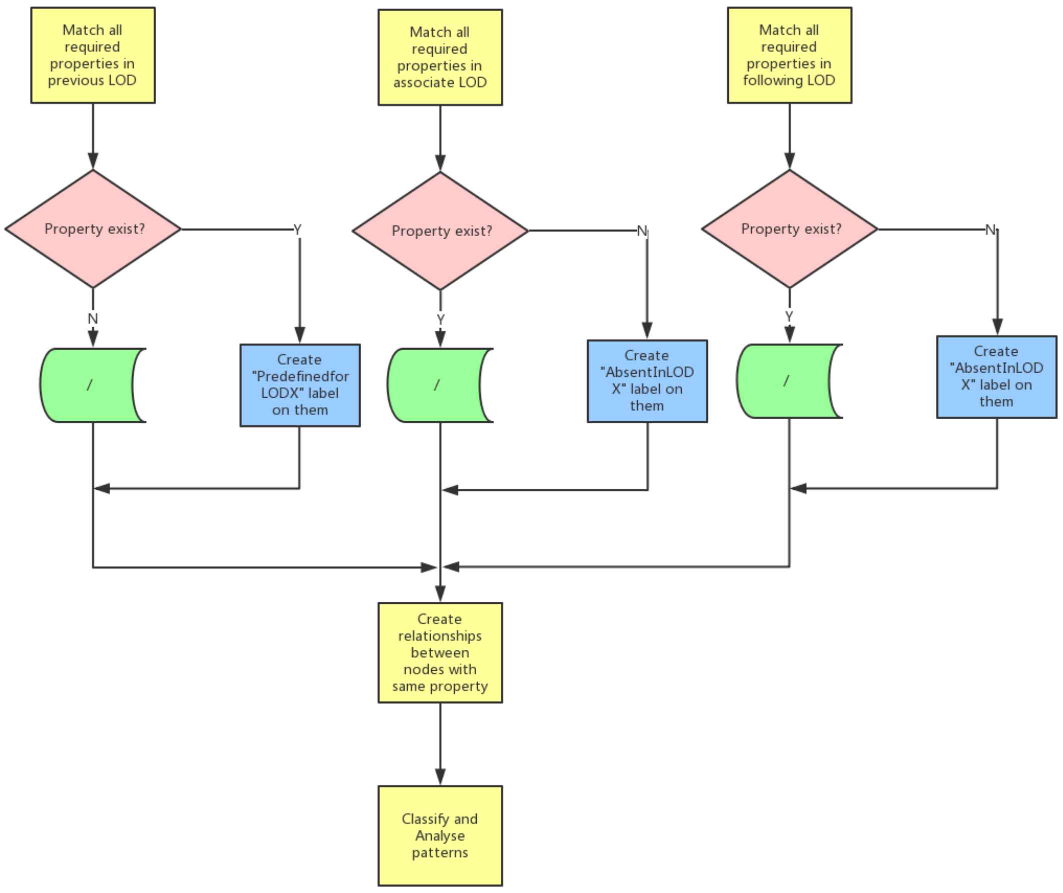

Match all required properties in previous LOD
Match all required properties in associate LOD
Match all required properties in following LOD
Property exist?
Property exist?
Property exist?
Y
N
N
N
Y
Y
/
/
/
Create "PredefinedforLODX" label on them
Create "AbsentInLODX" label on them
Create "AbsentInLODX" label on them
Create relationships between nodes with same property
Classify and Analyse patterns

Figure 4.16: Workflow Analysing Property Set

1. Correspondently merge the required property nodes in required LODs.
This step aims to identify missing properties in their required development stage. The nodes that are absent will be labelled as AbsentInLOD100/200/300. Different labels are assigned different colours, which can facilitate the finding of missing property nodes.

```
MERGE (a:LOD300{cls: "IfcPropertySingleValue",
Name:"LOD300 Property" })
ON CREATE SET a:AbsentInLOD300
```

Code Listing 4.11: Example of creating Property Node and setting the *AbsentInLODXXX* label if it does not exist in the required LOD

2. Correspondently merge the required property nodes from higher LODs in lower LODs.
The goal of this step is to find out missing properties by checking for their existence. The LOD is accumulative and should progress from LOD 100 at conceptual design through completion of construction(Bloomberg *et al.*, 2012). Therefore properties with lower LODs should be merged in all the higher LODs, because these properties are certainly a requirement of higher

LODs and can be absent. If they do not exist, then the property node should be created with the same name but labeled them with AbsentInLOD 100/200/300.

```
MERGE (a:LOD300{cls: "IfcPropertySingleValue",
Name:"LOD200 Property" })
ON CREATE SET a:AbsentInLOD300
```

Code Listing 4.12: Example of creating a Property Node and setting the *AbsentInLODXXX* label if it does not exist in the next LOD either

3. Correspondently match merge the required property nodes from higher LODs in lower LODs.
The goal of this step is to identify early defined properties by matching LODs. Properties from higher LODs should be checked in all lower LODs. If they are created earlier than demanded, they are marked with the label PredefinedforLOD100/200/300.

```
MATCH (a:LOD200{cls: "IfcPropertySingleValue",
Name:"LOD300 Property")
SET a:PredefinedforLOD300
```

Code Listing 4.13: Example of setting *PredefinedforLODXXX* Label on the Property Node that exists before the required LOD

4. Link the properties with the same name but different LODs together using the DevelopedTo relationship in order to show the inheritance relationship of all the properties, more exactly, both the existing properties and the merged properties. In this way, patterns about the evolution of property sets are generated for further queries.

```
MATCH (a:LOD200{cls: "IfcPropertySingleValue",
(b:LOD300{cls:"IfcPropertySingleValue"})
WHERE a <> b AND a.Name = b.Name
MERGE(a)-[r:DevelopsTo]-(b)
RETURN a,r,b
```

Code Listing 4.14: Example of connecting same Properties in different LODs with *DevelopsTo* Relationship

5. Read the graph and by distinguishing different patterns and colors, identify which properties are temporarily or always absent, which properties are created on time and how they change.

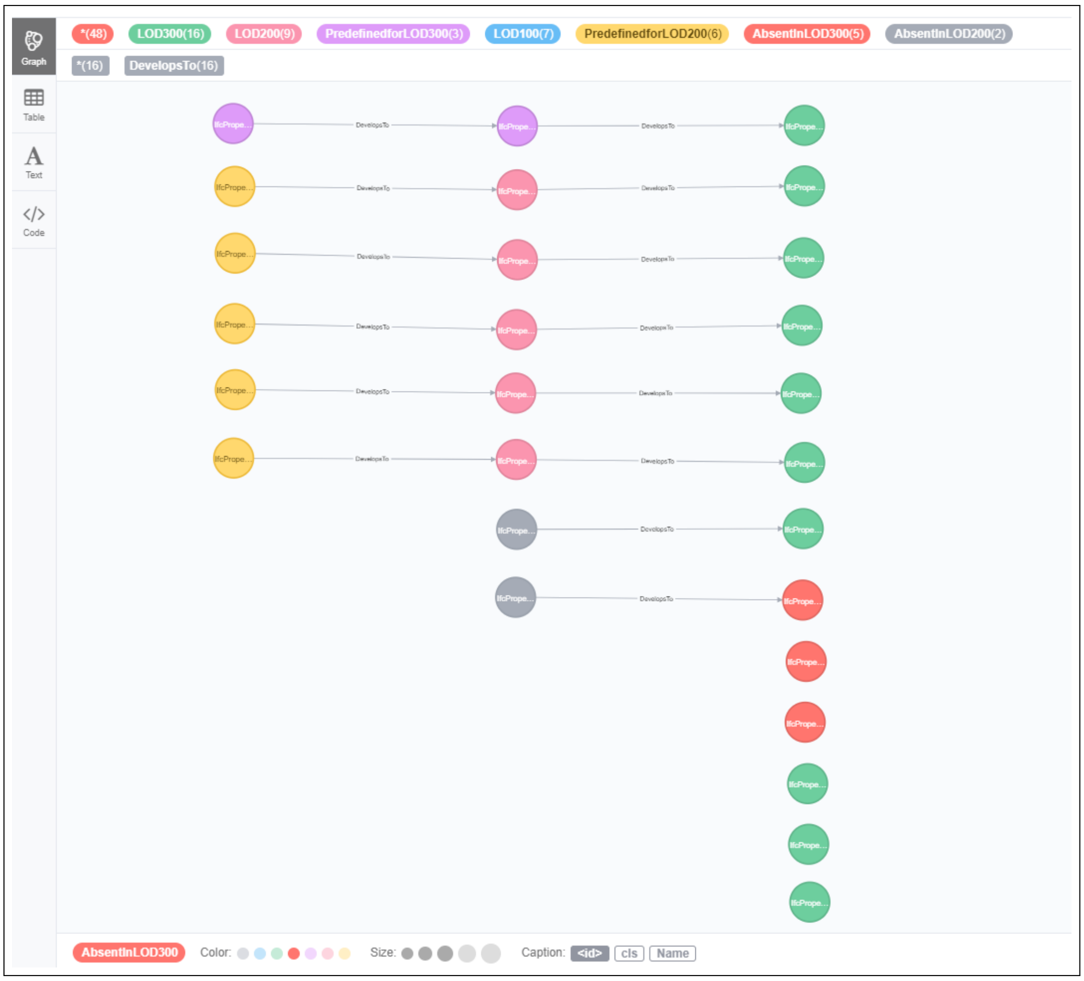


Figure 4.17: Final Result Property Set Query

4.2.5 Analyse Property Sets

In this graph, different patterns indicate different types of properties.
1. Set in the required LODs.

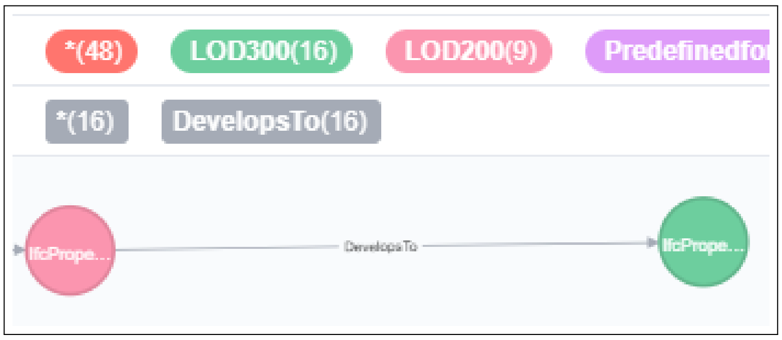


Figure 4.18: Pattern Type 1

The node colour of LOD300 and LOD200 indicate properties of type 1, which are set in the required LOD. The *DevelopedTo* connection shows an inheritance relationship.

2. Set in advanced before the required LOD.

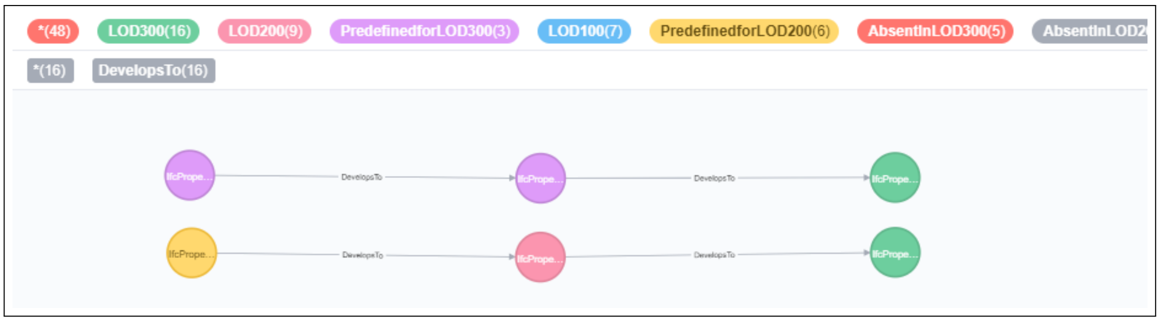


Figure 4.19: Pattern Type 2

The colours of the label PredefinedforLOD300 (purple) and PredefinedforLOD100 (yellow) imply that these properties are defined earlier as demanded. The first LOD300/LOD200 colour on the right shows in which LOD are they demanded.

3. Absent from the required LOD but defined later.

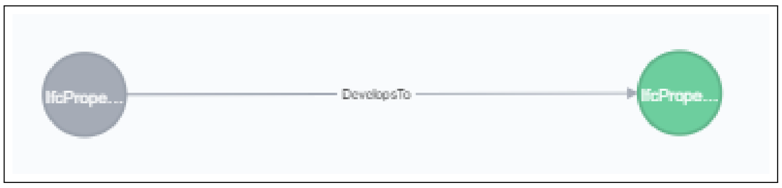


Figure 4.20: Pattern Type 3

This indicates properties that should be created in the grey LOD (colour for label AbsentIn-LOD200) but are absent until the green LOD (LOD300 colour).

4. Absent since the required LOD until the end.

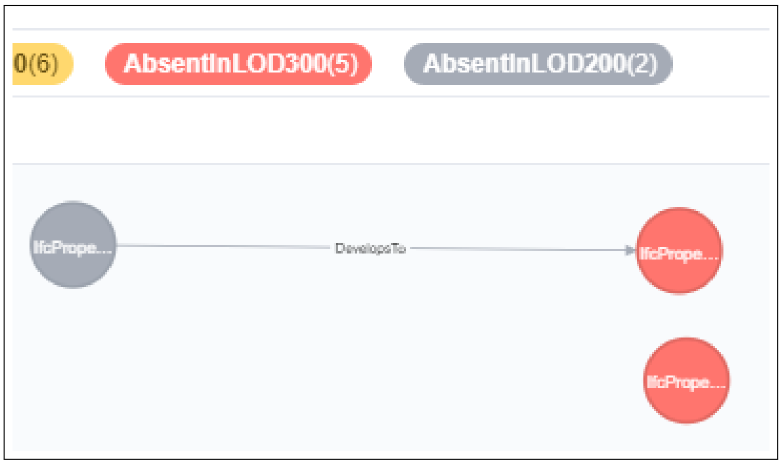


Figure 4.21: Pattern Type 4

This indicates properties that should be created in the grey LOD(colour for label AbsentIn-LOD200) on the left but that are absent until the red node (colour for label AbsentInLOD300) on the right. In the same way, the red node below implies a missing property in LOD 300.

Chapter 5

Model Checking with Graph Database

Besides checking the existence of data and the content of data about LOD, to utilize BIM it is also important that the starting point for downstream analysis is consistent. Furthermore, since BIM models consist of an amalgamation of models from different disciplines which may work in tandem, it is important to establish a well-coordinated approach. Hence, the predecessor of most of the objectives is the validation of the models that will be used analysis. Models should be consistent with regard to model structure, relationships and use of object classes/types and they should be checked for duplicates and intersections(Statsbygg, 2017).

5.1 BIM-Based Model Checking

BIM-based model checking is one of the benefits in using BIM. Comparing with traditional design method, it is much more convenient to spot flaws in BIM models. BIM based model checking involves compliance checking to individual BIM requirements and design coordinating. More exactly, it is not only about clash detections, but also about exploring its potential in solving new problems against various rule sets(Hjelseth, 2015). Rule-sets are collections of rules within one topic, such as BIM-validation (clash detection), space validation, model version comparison, comparing the structural versus architectural model and deficiency of components(Statsbygg, 2017). In commercial software Solibri, clash detection and deficiency detection are two main features of model checking. A clash occurs when elements of different systems occupy the same space or area. Deficiency detection is aimed to search for components and materials missing from the model. These two issues are mainly discussed in this chapter.

5.2 Analysis of Structural Consistency

Neo4j can also be applied to check the the deficiency of model structure. There are various rules defining deficiency issue. For example, columns should touch slabs, roofs, columns, or walls above or below themselves. In some cases, like in an ordinary frame structure, in which structural elements are spread uniformly, the vertical bearing structure is designed to be continuous from the roof to the ground. If the structure has numerous columns, it can occur that some are forgotten which will lead to errors when doing structural calculations. In Neo4j, the consistency of columns can be verified by querying the origin point of the columns. In short, if columns share same x-, y-coordinates and different, but yet continuous z-coordinates, it indicates that these columns are consecutive. Otherwise, the load-bearing path of the columns might be not continuous.

5.2.1 Retrieve Column Consistency List

First, a two-storey building with discontinuous columns is modelled. One column in the front corner is missing. The goal of the query is to generate a list regarding the column' position and storey number, which facilitates the finding of unusual information. Moreover, a graph representing the sequence and hierarchy of columns is helpful in locating missing elements.

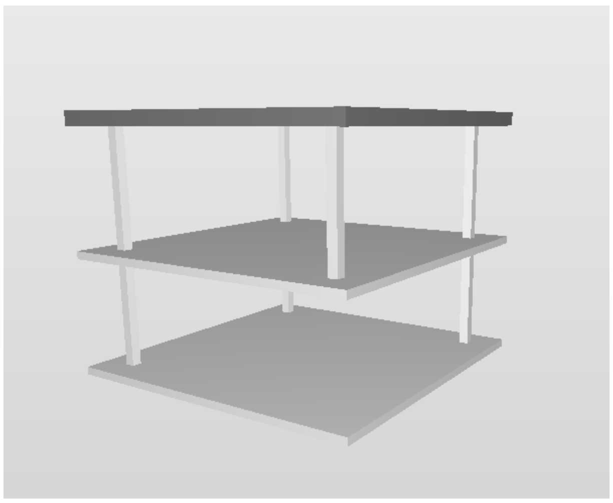
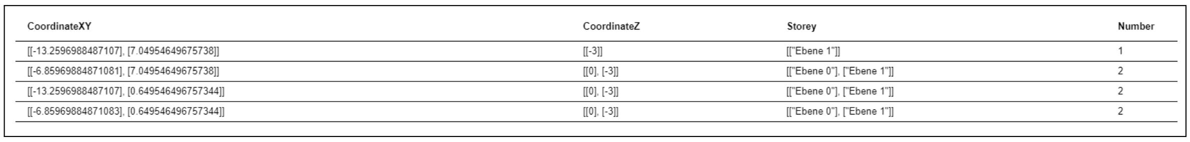


Figure 5.1: Building model for column consistency check

The coordinate of a column is given by *IfcCartesianPoint* which is associated with the object *IfcColumn*. With the help of the IFC objectified relationship *IfcRelContainedInSpatialStructure*, the column is then contained in the relevant *IfcBuildingStorey*. Therefore by querying this pattern, both coordinate and storey number of a certain column can be found.

Second, since columns in the same position will have the same x-, y-coordinates but different z-coordinates, it is essential to group them by their distinct x- and y-coordinates while collecting z-coordinates and storeys to categorize columns in the same place together. Then by counting column in the same location missing columns can be inferred.



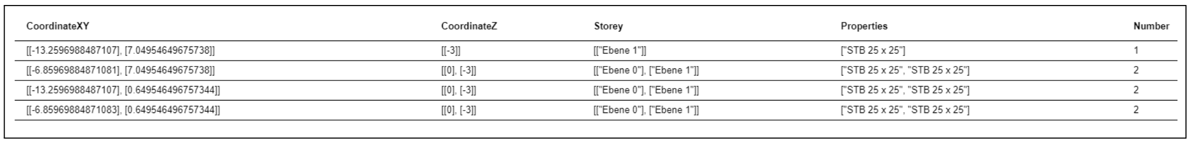
CoordinateXY	CoordinateZ	Storey	Number
[[-13.2596988487107], [7.04954649675738]]	[[-3]]	[["Ebene 1"]]	1
[[-6.85969884871081], [7.04954649675738]]	[[0], [-3]]	[["Ebene 0"], ["Ebene 1"]]	2
[[-13.2596988487107], [0.649546496757344]]	[[0], [-3]]	[["Ebene 0"], ["Ebene 1"]]	2
[[-6.85969884871083], [0.649546496757344]]	[[0], [-3]]	[["Ebene 0"], ["Ebene 1"]]	2

Figure 5.2: Result Column List Query

```
MATCH (d{cls:"IfcBuildingStorey"})-[]-
(c{cls:"IfcRelContainedInSpatialStructure"})-[]-
(b{cls:"IfcColumn"})-[*3]-(a{cls:"IfcCartesianPoint"})
WITH [[a.CoordinateX],[a.CoordinateY]] AS CoordinateXY,
[a.CoordinateZ] AS CoordinateZ,[d.LongName] AS Storey
RETURN distinct CoordinateXY, collect(CoordinateZ)
AS CoordinateZ, collect(Storey) AS Storey,
COUNT (distinct(CoordinateZ)) AS Number
ORDER BY COUNT (distinct(CoordinateZ))
```

Code Listing 5.1: Match and return column' x-, y- and z-Coordinates and Storey Information

From the list it is clear that around position (-13.26,7.05) only one column exist on the first storey while in other positions columns on both first storey and ground floor exist. The numbers, which are ordered in ascending order, indicate the conclusion. This list can be expanded to query the size, material and other properties too.



CoordinateXY	CoordinateZ	Storey	Properties	Number
[[-13.2596988487107], [7.04954649675738]]	[[-3]]	[["Ebene 1"]]	["STB 25 x 25"]	1
[[-6.85969884871081], [7.04954649675738]]	[[0], [-3]]	[["Ebene 0"], ["Ebene 1"]]	["STB 25 x 25", "STB 25 x 25"]	2
[[-13.2596988487107], [0.649546496757344]]	[[0], [-3]]	[["Ebene 0"], ["Ebene 1"]]	["STB 25 x 25", "STB 25 x 25"]	2
[[-6.85969884871083], [0.649546496757344]]	[[0], [-3]]	[["Ebene 0"], ["Ebene 1"]]	["STB 25 x 25", "STB 25 x 25"]	2

Figure 5.3: Column Consistency List with Properties

This is achieved by adding additional queries on desired properties.

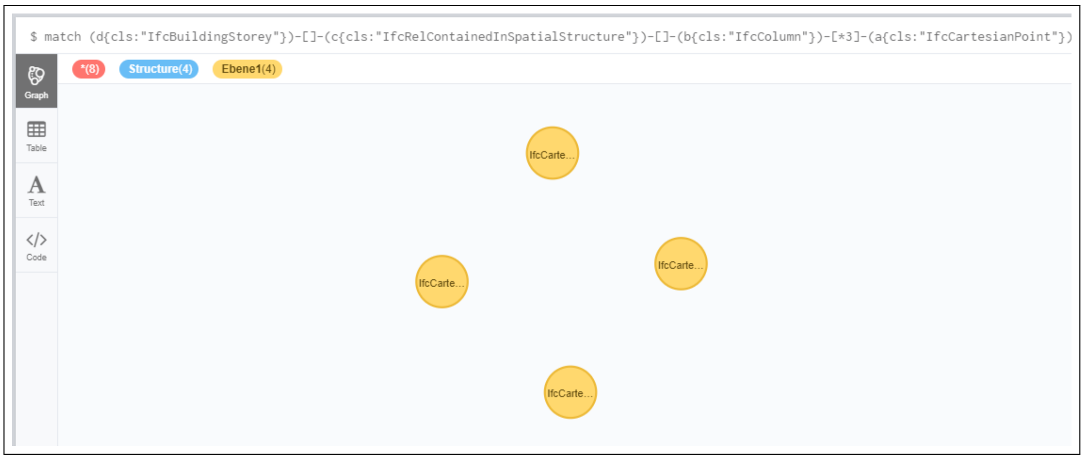
```
MATCH (d{cls:"IfcBuildingStorey"})-[]-
(c{cls:"IfcRelContainedInSpatialStructure"})-[]-
(b{cls:"IfcColumn"})-[*3]-(a{cls:"IfcCartesianPoint"})
WITH ...b.ObjectType AS Properties...
RETURN ...collect(Properties) AS Properties, ...
```

Code Listing 5.2: Match and return Columns' x-, y- and z-Coordinates, Storey Information and Properties

5.2.2 Retrieve Column Consistency Graph

To demonstrate the advantage of graph database in displaying complicated information in a clear and easily understandable way, Neo4j can generate graph about relationships of columns. However, because in IFC data, there is no explicit spatial definition about the continuity or intersection relationship between different shapes, which are not attached to spaces or other entities, such a definition needs to be created first.

The position and elevation of column are not directly described in *IfcColumn*. Instead, they are illustrated through relationship with other associate classes. Thus, the first step is to create labels about columns' storey number to directly distinguish them in different floors. This is done by using the *set:label* function on relevant Cartesian origin Cartesian points and *IfcColumns* themselves in Cypher.



```
$ match (d{cls:"IfcBuildingStorey"})-[]-(c{cls:"IfcRelContainedInSpatialStructure"})-[]-(b{cls:"IfcColumn"})-[*3]-(a{cls:"IfcCartesianPoint"})
```

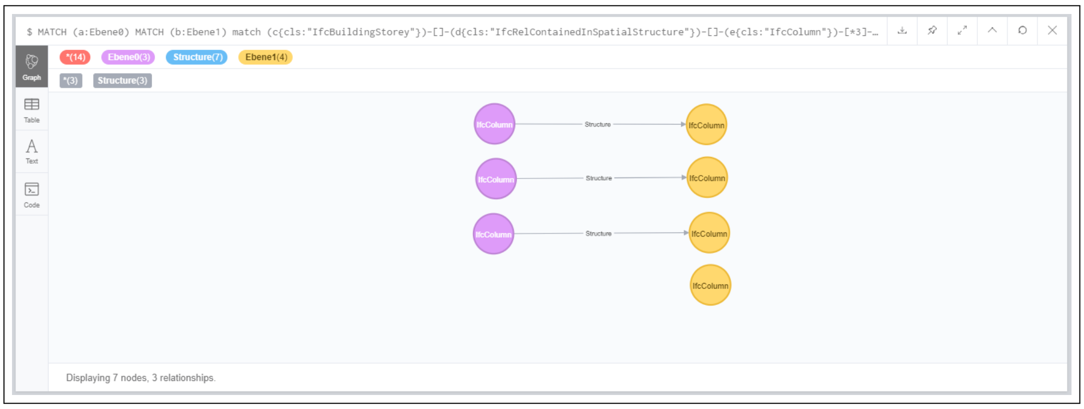
Figure 5.4: Setting "Ebene 1" labels on columns on the first floor

```
MATCH (d{cls:"IfcBuildingStorey"})-[]-
(c{cls:"IfcRelContainedInSpatialStructure"})-[]-
(b{cls:"IfcColumn"})-[*3]-(a\{cls:"IfcCartesianPoint"})
WHERE d.LongName="Ebene 1"
```

```
SET a:Ebene1
RETURN a
```

Code Listing 5.3: Match and set Storey Information on Columns as Labels

After adding storey information labels on all the columns and location nodes, the continuity or intersection relationship between them can be depicted by connecting columns with Cartesian points of the same x- and y-coordinate but different elevations. In this process the label and coordinate of these Cartesian points work as an index to classify and filter columns.



```
$ MATCH (a:Ebene0) MATCH (b:Ebene1) match (c{cls:"IfcBuildingStorey"})-[]-(d{cls:"IfcRelContainedInSpatialStructure"})-[]-(e{cls:"IfcColumn"})-[*3]-…
```

Displaying 7 nodes, 3 relationships.

Figure 5.5: Result Merging Column Continuity Relationship

```
MATCH (a:Ebene0)
MATCH (b:Ebene1)
MATCH (ccls:"IfcBuildingStorey")-[]-
   (cls:"IfcRelContainedInSpatialStructure")-[]-
(ecls:"IfcColumn")-[*3]-(acls:"IfcCartesianPoint")
MATCH (dcls:"IfcBuildingStorey")-[]-
   (cls:"IfcRelContainedInSpatialStructure")-[]-
(fcls:"IfcColumn")-[*3]-(bcls:"IfcCartesianPoint")
MATCH (cls:"IfcBuildingStorey")-[]-
   (cls:"IfcRelContainedInSpatialStructure")-[]-
(gcls:"IfcColumn")-[*3]-(cls:"IfcCartesianPoint")
WHERE a.CoordinateX = b.CoordinateX
AND a.CoordinateY = b.CoordinateY
AND a.CoordinateZ <> b.CoordinateZ
MERGE (e)-[:Structure]->(f)
RETURN e,f,g
```

Code Listing 5.4: Connecting Columns that have the same x- and y-Coordinates but different z-Coordinate with the *Structure* relationship

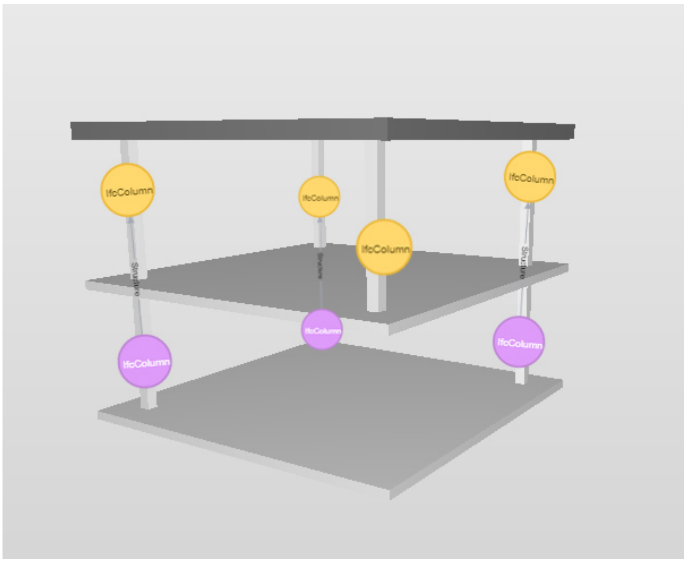


Figure 5.6: Building model and continuity relationship of columns

The connection between different nodes indicates continuous vertical bearing structure. With the ground floor label in purple and the first floor label in yellow, it is clear that the lonely yellow point stands for an unattached column below, which can be problematic in a framework structure. The utilization of the graph can broaden into checking the load-bearing path of the model structure and its continuity. For example the slab or eventually the beams can be included in the path.

5.3 Check Intersection Between Elements

Checking intersections, also known as clash detections, are aimed at resolving potential issues before construction on site and thus lowering down costs and time. Therefore, researching the capability of graph database in conducting clash detections can facilitate a good coordination in BIM. According to not only the Statsbygg BIM manual mentioned above, but also the Rijksgebouwendienst Building Information Model Standard from the Ministry of the Interior and Kingdom Relations of the Netherlands(Rgd BIM Norm), doubling and intersection are important part of general requirements of BIM extracts too. Doubling and intersection can occur due to drafting errors when multiple copies of a BIM object are modelled at the same location or from discipline specific detailing of building elements(Rijksgebouwendienst, 2012). They are generally not permitted and need to be examined before the model is delivered to other individuals.

5.3.1 Check Intersection Between the Same Line-Based Elements

There are various rules and standards regarding intersection definition. For walls in Revit, or more generally for the same building elements, it is usually not problematic if they touch each other, since in Revit they are automatically joined together in the contact area. However, if walls are collinear and overlap each other, it indicates that they are modelled in the same place twice and will be calculated twice. Therefore the decision processes are different for intersection between different elements and between the same elements. Within the same elements, overlapping needs to be specifically categorized besides point contact.

5.3.1.1 Retrieve Coordinate of the Wall

The location of the walls are stored in relevant IFC data. More specifically, walls are exchanged by an IFC2× file as instances of *IfcWall* or *IfcWallStandardCase*. Here since walls have single material thickness they are defined by *IfcWallStandardCase*, which gives the body and axis information. Because little explicit shape contact information, which is not attached to spaces or other entities, is defined in IFC, retrieving spatial intersection is sophisticated without an external geometry engine. Such engines can create geometry and interpret the form with their own algorithm. The interpretations of the engine can possibly be reimported into graph database for further queries. However to reduce the complexity of the task and focus on displaying the capability of Neo4j, the task is concentrated into checking the wall axis line segment intersections with only graph databases.

The origin of the wall axis lies in a Cartesian point, which has the following connection with the wall: *IfcWallStandardCase* - *IfcLocalPlacement* - *IfcAxis2Placement3D* - *IfcCartesian-Point*. The end point is not given. To find out the end point, the next step is to discover the axis. The wall axis,which is given by an instance of IfcShapeRepresentation with the following conventions as a straight wall(Liebich, 2009):

IfcShapeRepresentation.RepresentationIdentifier = ”Axis”
IfcShapeRepresentation.RepresentationType = ”Curve2D”

The geometric representation item is indicated by IfcShapeRepresentation. For straight walls in this model, the item is an IfcPolyline with exactly two points. One is $(0, 0)$ and the other is $(x, 0)$(x is a placeholder for the x-coordinate), indicating the length of the wall.

Secondly, the directions of the IfcPolylines are given by class *IfcDirection* with relationship *IfcWallStandardcase* - *IfcLocalPlacement* - *IfcAxis2Placement3D* - *IfcDirection* with the wall. In this case the walls go along the y- axis and therefore the direction vector is IfcDirection$((0., 1., 0.))$. For the nonorthogonal axes, the direction is represented by a vector $(x., y., 0.)$ with x and y, as coordinate placeholder, if the elevation of the wall axis remains the same.

With the original point, length and direction of the wall, the coordinate of the end point can be inferred. If that the origin coordinate is (a, b, c), the length is x and direction vector is $(d, e, 0)$, the coordinate of the end point is:

$$(a + x \times cos(\alpha), b + x \times sin(\alpha), c)$$

$$with\ cos(\alpha) = \frac{d}{\sqrt{d^2 + e^2}}, sin(\alpha) = \frac{e}{\sqrt{d^2 + e^2}}$$

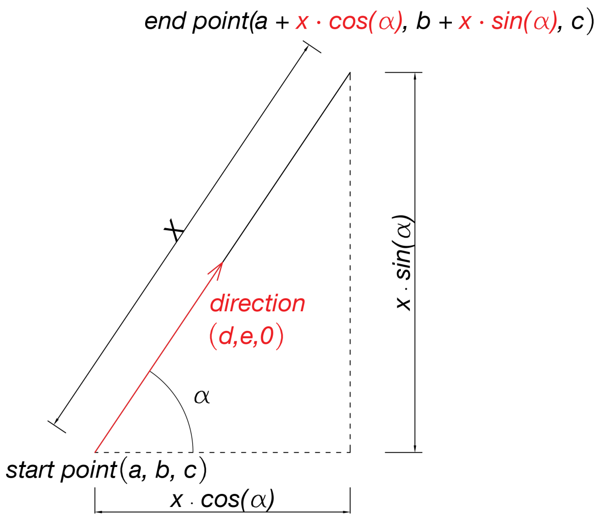


Figure 5.7: End point calculation of wall axis

```
MATCH (a{cls:"IfcCartesianPoint"})-[]-
(e{cls:"IfcAxis2Placement3D"})-[]-({cls:"IfcLocalPlacement"})
-[]-(i{cls:"IfcWallStandardCase"})-[]-
({cls:"IfcProductDefinitionShape"})-[]-
({cls:"IfcShapeRepresentation",RepresentationType:"Curve2D",
RepresentationIdentifier: "Axis"})-[]-({cls:"IfcPolyline"})
-[]-(b{cls:"IfcCartesianPoint"}),
(e{cls:"IfcAxis2Placement3D"})-[]-(f{cls:"IfcDirection"}),
(c{cls:"IfcCartesianPoint"})-[]-(g{cls:"IfcAxis2Placement3D"})
-[]-({cls:"IfcLocalPlacement"})-[]-
(l{cls:"IfcWallStandardCase"})-[]-
({cls:"IfcProductDefinitionShape"})-[]-
({cls:"IfcShapeRepresentation",RepresentationType:"Curve2D",
RepresentationIdentifier:"Axis"})-[]-({cls:"IfcPolyline"})-[]-
(d{cls:"IfcCartesianPoint"}),
(g{cls:"IfcAxis2Placement3D"})-[]-(h{cls:"IfcDirection"})
WHERE (b.CoordinateX<>0 OR b.CoordinateY<>0)
```

```
AND (d.CoordinateX<>0 OR d.CoordinateY<>0)
AND f.nid<>19 AND h.nid<>19 AND i.Name <> l.Name
WITH a.CoordinateX + b.CoordinateX*f.CoordinateX/
sqrt(f.CoordinateX\^{}2+f.CoordinateY\^{}2) as AS,
a.CoordinateX + b.CoordinateX*f.CoordinateX/
sqrt(f.CoordinateX\^{}2+f.CoordinateY\^{}2) as AS,
a.CoordinateY + b.CoordinateX*f.CoordinateY/
sqrt(f.CoordinateX\^{}2+f.CoordinateY\^{}2) AS y2,
a.CoordinateX AS x1,a.CoordinateY AS y1,
c.CoordinateX + d.CoordinateX*h.CoordinateX/
sqrt(h.CoordinateX\^{}2+h.CoordinateY\^{}2) AS x4,
c.CoordinateY+d.CoordinateX*h.CoordinateY/
sqrt(h.CoordinateX\^{}2+h.CoordinateY\^{}2) AS y4,
c.CoordinateX AS x3,c.CoordinateY AS y3,
i.Name AS Wall1, l.Name AS Wall2,
```

Code Listing 5.5: Match Wall origin Coordinates, Length and Axis Direction, calculate Coordinates of the two end Points of Wall Axis

In the syntax, the start point of the first wall axis is set as (x_1, y_1) and end point as (x_2, y_2), assuming that the z-coordinate is constant. In the same way the second wall axis is defined as (x_3, y_3) and (x_4, y_4).

5.3.1.2 Line Segment Intersection Detection

With the coordinates of two wall axis (or other equivalent linebased building elements), the task turns into a geometry problem detecting intersection of two line segments in the xy plane.

Firstly, it is to make sure that bounding boxes of both lines need to overlap if the lines should intersect. Bounding boxes are boxes around line segments such that endpoints of both line segments are at the corner of the boxes and edges of the boxes are in parallel to the coordinate axes. In logic, overlapping bounding box is necessary to intersection of line segments.

$$\textit{bounding box overlap} \rightarrow \textit{line segments intersect}$$

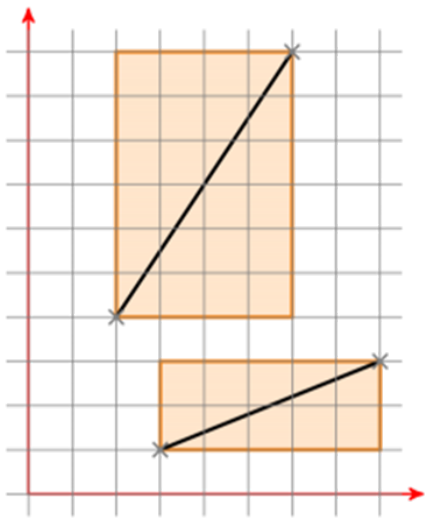


Figure 5.8: Bounding boxes of two Lines(Thoma, 2013)

This is done by comparing the x coordinate and y coordinate of both line segments. If any of the following condition is met, it is not possible for the lines to meet:

1. The maximal x value of line A is smaller than minimal x value of line B
2. The maximal x value of line B is smaller than minimal x value of line A
3. The maximal y value of line A is smaller than minimal y value of line B
4. The maximal y value of line B is smaller than minimal y value of line A

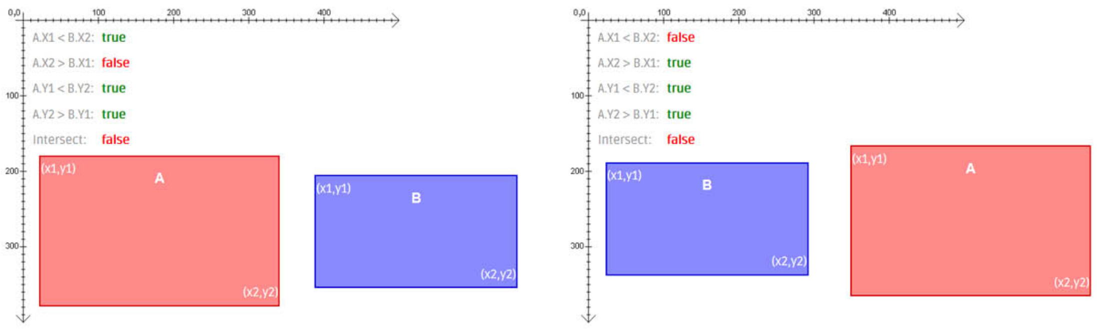


Figure 5.9: Left: condition 1, Right: condition 2

3. The maximal y value of line A is smaller than minimal y value of line B
4. The maximal y value of line B is smaller than minimal y value of line A

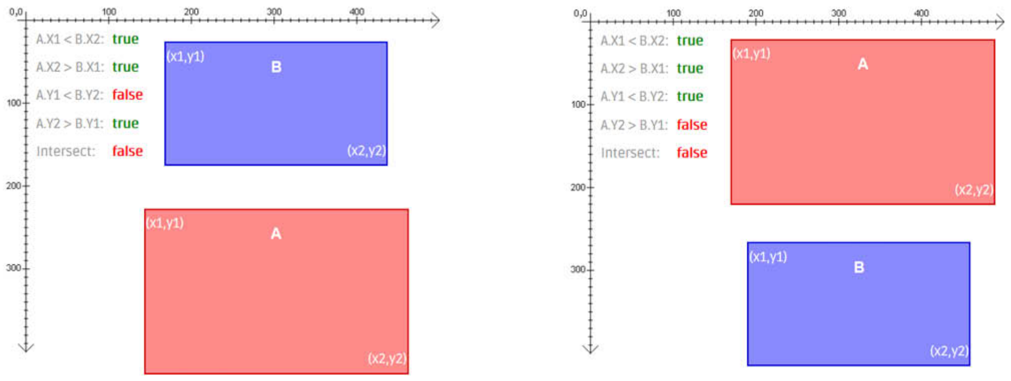


Figure 5.10: Left: condition 3, Right: condition 4

Not meeting all these four criteria is the necessary condition of intersection, but it does not definitely lead to intersection.

Secondly, two lines need to "cross" each other if they intersect. This is indicated by the cross product of both lines. If $\overrightarrow{CA} \times \overrightarrow{CD}$ and $\overrightarrow{CB} \times \overrightarrow{CD}$ have the opposite sign, then A and B should be on the different sides of line CD. Therefore $(\overrightarrow{CA} \times \overrightarrow{CD}) \cdot (\overrightarrow{CB} \times \overrightarrow{CD}) < 0$. One special condition is that A or B is on line CD. In this case line AB and CD also intersect and $(\overrightarrow{CA} \times \overrightarrow{CD}) \cdot (\overrightarrow{CB} \times \overrightarrow{CD}) = 0$. In a word $(\overrightarrow{CA} \times \overrightarrow{CD}) \cdot (\overrightarrow{CB} \times \overrightarrow{CD}) \leq 0$ if they intersect or are collinear. Vice versa, $(\overrightarrow{AD} \times \overrightarrow{AB}) \cdot (\overrightarrow{AC} \times \overrightarrow{AB}) \leq 0$ if C, D "cross" line segment AB or on line AB. In combination with the condition that the bounding box of two lines segments overlap, two line segments must intersect or overlap.

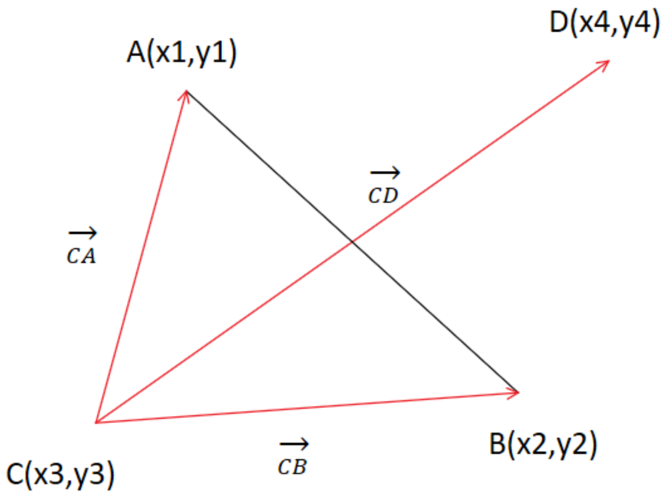


Figure 5.11: Situation when A, B "cross" line CD

Vector $\overrightarrow{CA}$ is $(x_1 - x_3, y_1 - y_3)$ while vector $\overrightarrow{CD}$ is $(x_4 - x_3, y_4 - y_3)$. For $\overrightarrow{CA}$ and $\overrightarrow{CD} \in \mathbb{R}^2$, the value of their cross product: $\overrightarrow{CA} \times \overrightarrow{CD} \to \mathbb{R}$ can be defined as :

$$(\overrightarrow{CA} \times \overrightarrow{CD}) = det(\overrightarrow{CA}, \overrightarrow{CD}) = det \begin{bmatrix} x_1 - x_3 & x_4 - x_3 \\ y_1 - y_3 & y_4 - y_3 \end{bmatrix}$$

$$= (x_1 - x_3) \cdot (y_4 - y_3) – (x_4 - x_3) \cdot (y_1 - y_3).$$

Similarly value of $\overrightarrow{CB} \times \overrightarrow{CD}$ equals

$$(x_2 - x_3) \cdot (y_4 - y_3) - (y_2 - y_3) \cdot (x_4 - x_3),$$

$\overrightarrow{AD} \times \overrightarrow{AB}$ equals

$$(x_4 - x_1) \cdot (y_2 - y_1) - (y_4 - y_1) \cdot (x_2 - x_1),$$

$\overrightarrow{AC} \times \overrightarrow{AB}$ equals

$$(x_3 - x_1) \cdot (y_2 - y_1) - (y_3 - y_1) \cdot (x_2 - x_1).$$

If anyone in $(\overrightarrow{CA} \times \overrightarrow{CD}) \cdot (\overrightarrow{CB} \times \overrightarrow{CD})$ or $(\overrightarrow{AD} \times \overrightarrow{AB}) \cdot (\overrightarrow{AC} \times \overrightarrow{AB})$ is positive, it implies that at least one line dose not "cross" the other one. Thus two line segments don't intersect. In logic the decision can be represented as:

$$\textit{Lines not "cross" each other} \to \textit{Line segments do not intersect}$$

```
(c.CoordinateX + d.CoordinateX*h.CoordinateX/
sqrt(h.CoordinateX\^{}2+h.CoordinateY\^{}2)
 - a.CoordinateX)*(a.CoordinateY +
 b.CoordinateX*f.CoordinateY/sqrt(f.CoordinateX\^{}2
 + f.CoordinateY\^{}2)- a.CoordinateY) -
(c.CoordinateY+d.CoordinateX*h.CoordinateY/
sqrt(h.CoordinateX\^{}2 + h.CoordinateY\^{}2) -
a.CoordinateY)*(a.CoordinateX + b.CoordinateX*f.CoordinateX/
sqrt(f.CoordinateX\^{}2 + f.CoordinateY\^{}2) -
   a.CoordinateX) AS num1,
(c.CoordinateX - a.CoordinateX)*(a.CoordinateY +
b.CoordinateX*f.CoordinateY/sqrt(f.CoordinateX\^{}2 +
   f.CoordinateY\^{}2)
- a.CoordinateY) - (c.CoordinateY  - a.CoordinateY)
*(a.CoordinateX + b.CoordinateX*f.CoordinateX/
sqrt(f.CoordinateX\^{}2 + f.CoordinateY\^{}2) -
   a.CoordinateX) AS num2,
(a.CoordinateX - c.CoordinateX)*(c.CoordinateY +
   d.CoordinateX*
h.CoordinateY/sqrt(h.CoordinateX\^{}2 + h.CoordinateY\^{}2) -
c.CoordinateY) - (a.CoordinateY - c.CoordinateY)*
(c.CoordinateX + d.CoordinateX*h.CoordinateX/
sqrt(h.CoordinateX\^{}2 + h.CoordinateY\^{}2) -
   c.CoordinateX) AS num3,
(a.CoordinateX + b.CoordinateX*f.CoordinateX/
sqrt(f.CoordinateX\^{}2 + f.CoordinateY\^{}2) -
c.CoordinateX)*(c.CoordinateY + d.CoordinateX*h.CoordinateY/
sqrt(h.CoordinateX\^{}2 + h.CoordinateY\^{}2) -
   c.CoordinateY) -
(a.CoordinateY + b.CoordinateX*f.CoordinateY/
sqrt(f.CoordinateX\^{}2 + f.CoordinateY\^{}2) -
   c.CoordinateY)*
(c.CoordinateX + d.CoordinateX*h.CoordinateX/
sqrt(h.CoordinateX\^{}2 + h.CoordinateY\^{}2) -
   c.CoordinateX) AS num4
UNWIND [x1,x2] as line1x UNWIND [x3,x4] as line2x
UNWIND [y1,y2] as line1y UNWIND [y3,y4] as line2y
```

Code Listing 5.6: Calculate Values of four cross Products with end Point Coordinates of Wall Axis, which are stored in Rows in the end

Value of $\overrightarrow{CA} \times \overrightarrow{CD}$, $\overrightarrow{CB} \times \overrightarrow{CD}$, $\overrightarrow{AD} \times \overrightarrow{AB}$ and $\overrightarrow{AC} \times \overrightarrow{AB}$ are defined as num1, num2, num3 and num4. X coordinates of line AB and CD are stored in row *line1x* and *line2x*. Y coordinates of line AB and CD are stored in row *line1y* and *line2y*.

The general algorithm of the syntax is, firstly check if the bounding boxes of two line segments overlap. When not, it is not possible for the lines to meet. If yes, calculate the product of cross product pairs $\overrightarrow{CA} \times \overrightarrow{CD}$, $\overrightarrow{CB} \times \overrightarrow{CD}$, $\overrightarrow{AD} \times \overrightarrow{AB}$ and $\overrightarrow{AC} \times \overrightarrow{AB}$ to find out whether both lines "cross" each other. Only if the end points of both lines are at both sides of other line, the two lines can intersect.

However as mentioned in the beginning, overlapping is generally more problematic than intersection between same element type. Therefore overlap needs to be specifically detected and listed.

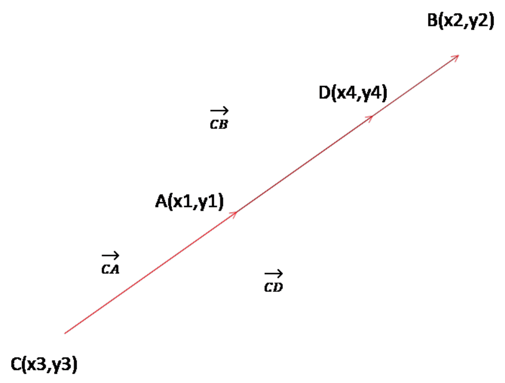


Figure 5.12: Situation when line AB intersect line CD

If lines overlap, then all $\overrightarrow{CA} \times \overrightarrow{CD}$, $\overrightarrow{CB} \times \overrightarrow{CD}$, $\overrightarrow{AD} \times \overrightarrow{AB}$ and $\overrightarrow{AC} \times \overrightarrow{AB}$ are 0, taken that overlapping of their bounding boxes is also fulfilled. One exception is that if they are collinear and have one end point in common. This will be excluded by making sure that they don't have points which shares the same x and y coordinates. With the new decision process, The whole algorithm becomes:

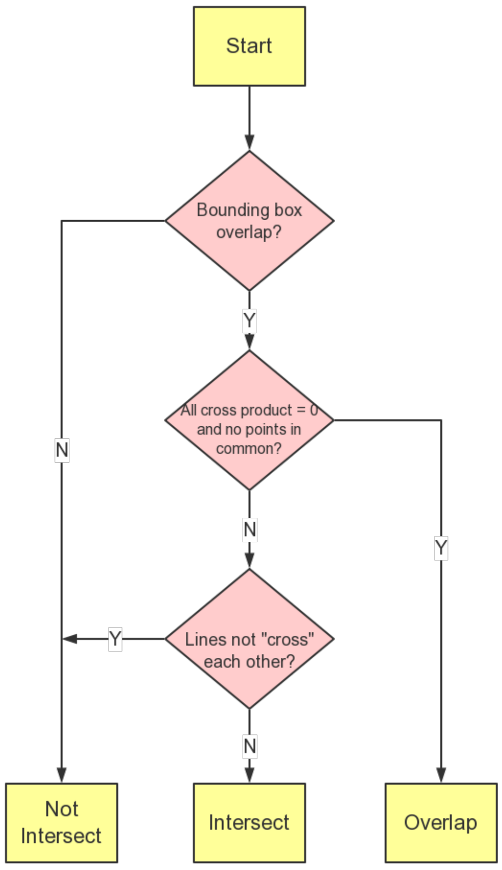
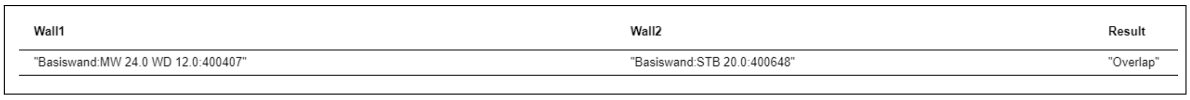

Start
Bounding box overlap?
Y
N
All cross product = 0 and no points in common?
N
Y
Lines not "cross" each other?
Y
N
Not Intersect
Intersect
Overlap

Figure 5.13: Algorithm of deciding whether two line segments intersect or overlap

The decision syntax is integrated at the end by the *RETURN* part. With the collected information the program will output the detection result of whether two lines segments intersect, not intersect or overlap.

```
RETURN Wall1, Wall2, CASE WHEN
max(line1x)<min(line2x) OR max(line2x)<min(line1x) OR
max(line1y)<min(line2y) OR max(line2y)<min(line1y)
\\Bounding box dectection
THEN "Not Intersect"
WHEN max(line1x)>=min(line2x) AND max(line2x)>=min(line1x) AND
max(line1y)>=min(line2y) AND max(line2y)>=min(line1y) AND
num1=0 AND num2=0 AND num3=0 AND num4=0 AND
(NOT (x1=x3 AND y1=y3) OR (x1=x4 AND y1=y4) OR (x2=x3
AND y2=y3) OR (x2=x4 AND y2=y4))
\\No common points
THEN "Overlap"
\\All cross product = 0
WHEN max(line1x)>=min(line2x) AND max(line2x)>=min(line1x) AND
max(line1y)>=min(line2y) AND max(line2y)>=min(line1y) AND
(num1*num2>0 OR num3*num4>0)
THEN "Not Intersect"
\\Any product of cross product pair > 0
ELSE "Intersect" END AS Result SKIP 1
\\Bounding box overlap and lines cross each other
```

Code Listing 5.7: Decision Statement which consists of Boundary Box Check, common Points Detection and Product of cross Product Check to distinguish if two Line Segments Intersect, Not Intersect or Overlap



Wall1	Wall2	Result
"Basiswand:MW 24.0 WD 12.0:400407"	"Basiswand:STB 20.0:400648"	"Overlap"

Figure 5.14: Result Wall Intersection Query

The result shows that the masonry wall with thickness of two layers as 24cm and 12cm overlaps the concrete wall with thickness 20cm. Materials and room boundaries are considered twice there. It may lead to overestimation of quantity take-offs or wrong area calculation. This result is in accordance with the intersection check of walls in Solibri, which is illustrated in Figure 5.15. In the future application, it can be used to detect and point out all the overlapped walls in a more complicated building by displaying them in an user interface.

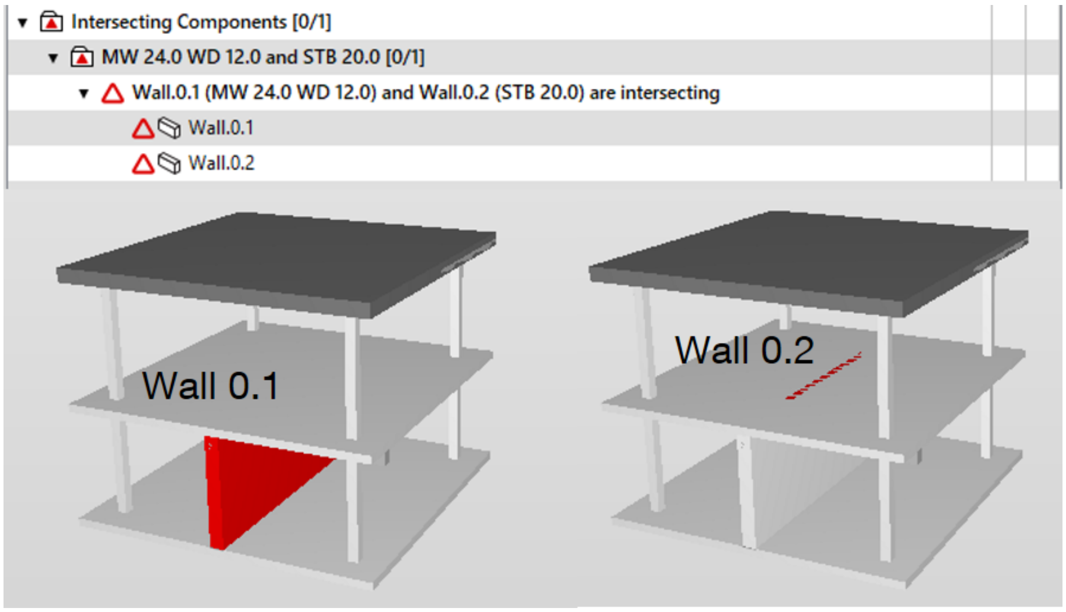

Intersecting Components [0/1]
MW 24.0 WD 12.0 and STB 20.0 [0/1]
Wall.0.1 (MW 24.0 WD 12.0) and Wall.0.2 (STB 20.0) are intersecting
Wall.0.1
Wall.0.2

Figure 5.15: Wall Intersection Check Result in Solibri and Position of overlapped Walls

One possible obstacle to doing an intersection check in Neo4j is that it does not support many spatial and mathematical functions, like Boolean operation or matrix calculations. Much effort should be made in writing spatial queries through basic functions. Moreover, IFC data does not directly indicate whether two shapes "touch" each other if they are not attached to spaces or other connected objects, which makes it complicate to detect it without regenerating the geometry. The advantage of graph databases lies in its ability to filter and retrieve data by patterns, labels and properties of the nodes. This enables a quick and precise search of desired information, such as looking for brick thickness of masonry walls whose orientation are in the y-axis of the global coordinate and located on the ground floor which contains concrete walls, by looking for *IfcMaterialLayer* of *IfcWalls* with their axis in the form of *IfcDirection* as (0,1,0) and their associate *IfcBuildingStorey* holding other concrete *IfcWalls*.

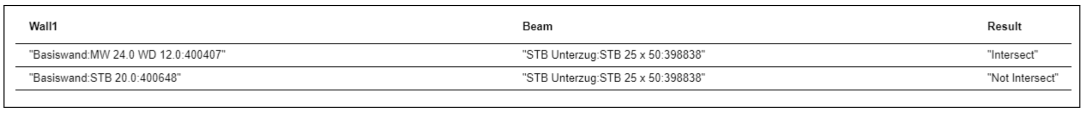
5.3.2 Check Intersection Between Different Line-Based Elements

The next attempt is to check intersection between different line-based Elements. One of the most practical and significant example is intersection between walls and beams. For walls and beams, the difference of the syntax lies in changing matched instances from two *IfcWall-Standardcases* to one *IfcWallStandardcase* and one *IfcBeam*. The description of the original point, direction and length of the beam in the model has the same structure as the description of walls. In other cases the geometry description may vary and the pattern filter should adapt to it accordingly.

```
From MATCH...-[]-(i{cls:"IfcWallStandardCase"})-[]-...
          ...-[]-(l{cls:"IfcWallStandardCase"})-[]-...

 To MATCH...-[]-(i{cls:"IfcWallStandardCase"})-[]-...
          ...-[]-(l{cls:"IfcBeam"})-[]-...
```

Code Listing 5.8: Change of Match Syntax from finding two Walls to finding one Wall and one Beam

It is assumed that the walls reach the floor slab, to which the beam is attached to ensure intersection when their 2D-axes meet. One difficulty is that when the axis lies in the default direction $(1, 0, 0)$, IfcDirection will be omitted and cannot be detected. Therefore, in order to retrieve the direction information in the program, it needs to be created manually or automatically given using external functions like the function *apoc.do.when* (distribute direction vector $(1, 0, 0)$ when not detected) in APOC plugin, which is a package of procedures for Neo4j. Because Cypher itself does not support full-blown conditional statements, as the *CASE* function is limited to returning a literal expression. In other words, statements like "*if a>0 then set b=1 else set c=1*" cannot be directly expressed. Furthermore, the geometry parser can be improved to automatically supplement the omitted information when the axis lies in $(1, 0, 0)$. To simply demonstrate Neo4j' query ability, the omitted direction data is added manually.



Wall1	Beam	Result
"Basiswand:MW 24.0 WD 12.0:400407"	"STB Unterzug:STB 25 x 50:398838"	"Intersect"
"Basiswand:STB 20.0:400648"	"STB Unterzug:STB 25 x 50:398838"	"Not Intersect"

Figure 5.16: Result Intersection Query between Walls and Beams

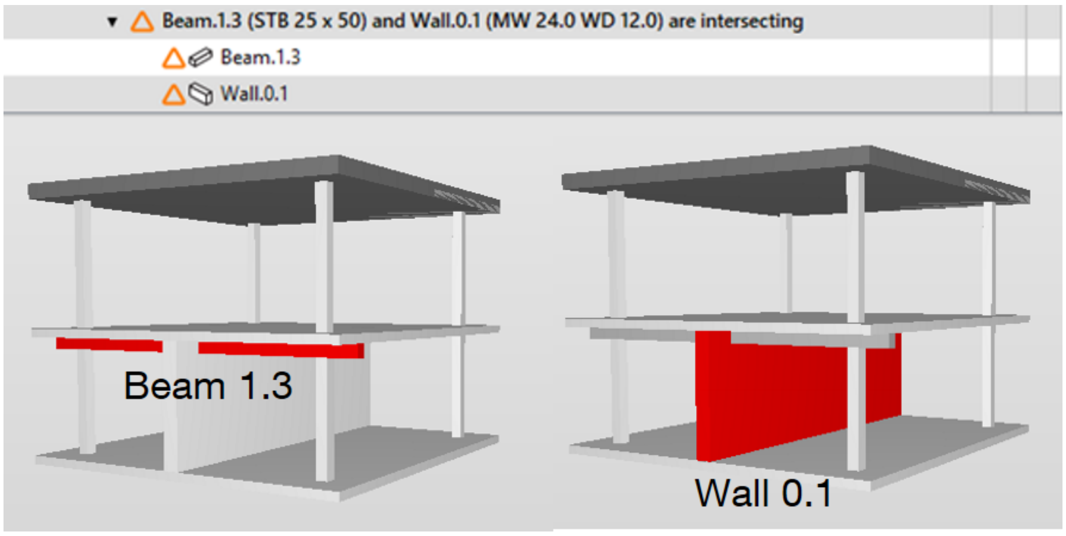

Beam.1.3 (STB 25 x 50) and Wall.0.1 (MW 24.0 WD 12.0) are intersecting
Beam.1.3
Wall.0.1

Figure 5.17: Wall and Beam Intersection Check Result in Solibri and Position of the intersected Beam and the Wall

The above figure shows the check result between all the beams and walls. The second line indicates that a beam with size 25 × 50 intersects wall with one 24cm-layer and one 12cm-layer, while the first line shows that it does not touch the wall made of 20cm thick reinforced concrete. This is in accordance with the result of Solibri shown in Figure 5.17. Consideration needs to be taken to fix the clash between the beam 398838 and wall 400407, such as combining them in Revit.

5.3.3 Check Intersection Between Face-Based Elements

Clash between slabs is also an important part of the structural check in the Solibri model checker. In the architectural, engineering, construction, owner/operator (AECO) field, overlapping can occur owing to negligence in modelling and numerous horizontal face-based elements like structural floor, hollow floor and suspended ceiling and roof, thus fulfilling various requirements of different disciplines like fire protection, sound isolation, architecture and construction.

There can be numerous situations and rules regarding slab intersection. In this demonstration, taking most frequent use cases into consideration, it is assumed that slabs have constant thickness, that they are rectangles and that their length and width are parallel to the x and y axes of global coordinates.

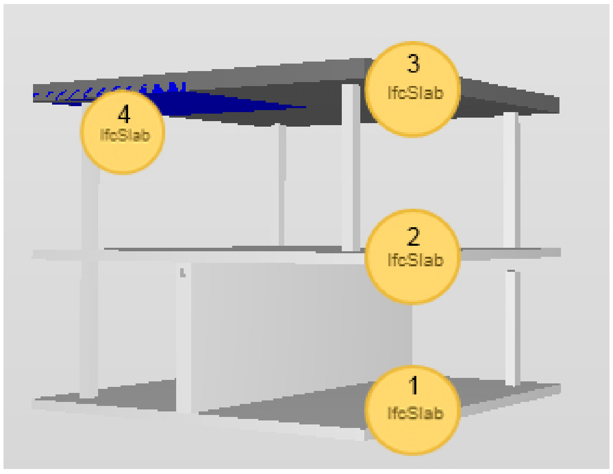


Figure 5.18: Position of Slabs in the Building Model

5.3.3.1 Retrieve Coordinate of the Slab

The global z-coordinate of the slabs are not indicated directly in the slab property. Instead, their elevations are defined by the building storey class to which they are connected. Because the coordinates of slabs do not contain elevation information, in a multi-storey building, it is essential to ensure that slabs in the clash check are in the same storey. The reference of building storey is different between roof slabs and floor slabs. For roofs, the relationship is *IfcSlab* <- *IfcRelAggregates* -> *IfcRoof* <- *IfcRelContainedInSpatialStructure* -> *IfcBuildingStorey*. In contrast, connections for floor slabs are *IfcSlab* <- *IfcRelContainedInSpatialStructure* -> *IfcBuildingStorey*.

After interpreting the storey of the slabs, the next step is to discover the coordinates of the corners of the roof and its length and width. Given that the slabs are rectangles with constant thickness, their location can be described by one center/corner point in global coordinate and their side length in the x, y direction.

The geometric representation of *IfcSlab* is given by the *IfcProductDefinitionShape*, allowing multiple geometric representations. Among them are local placement and geometric representations. *IfcLocalPlacement* defines the local coordinate system that is referenced by all geometric representations(Liebich, 2009). *IfcShapeRepresentation* holds the "SweptSolid" geometric representation, which is accompanied by *IfcExtrudedAreaSolid* and *IfcRectangleProfileDef* to indicate its depth and profile. The *IfcRectangleProfileDef* defines a rectangle by its X extent and its Y extent, and is placed within the 2D position coordinate system. In the model it is placed centrically within the position coordinate system, since the internally referenced x, y offsets are 0.

Floor slabs and roof slabs have different geometric and positional representations. For roof slabs, absolute placement is defined by *IfcLocalPlacement* within the world coordinate system.

Unlike floor slabs, the absolute placement is located in the lower left corner of the rectangle. Because the original coordinate needs to be placed centrically within the position coordinate system for *IfcRectangleProfileDef* with x,y offset as 0, the original coordinate needs to be placed centric within position coordinate system, the original point of the roof slab rectangle is offset to the center by using a position location addition $(length/2, width/2, 0)$.

```
MATCH
(storey{cls:"IfcBuildingStorey"})<-[]-
({cls:"IfcRelContainedInSpatialStructure"})-[]->
({cls:"IfcRoof"})<-[]-({cls:"IfcRelAggregates"})-[]->
(roofslab1{cls:"IfcSlab"})-[]-({cls:"IfcLocalPlacement"})-[]-
({cls:"IfcLocalPlacement"})-[]-({cls:"IfcAxis2Placement3D"})
-[]-(origin1{cls:"IfcCartesianPoint"}),
(roofslab1{cls:"IfcSlab"})-[]-
({cls:"IfcProductDefinitionShape"})
-[]-({cls:"IfcShapeRepresentation"})-[]-
({cls:"IfcExtrudedAreaSolid"})-[]-
(pro1{cls:"IfcRectangleProfileDef"}),
({cls:"IfcRoof"})<-[]-({cls:"IfcRelAggregates"})-[]->
(roofslab2{cls:"IfcSlab"})-[]-({cls:"IfcLocalPlacement"})-[]-
({cls:"IfcLocalPlacement"})-[]-({cls:"IfcAxis2Placement3D"})
-[]-(origin2{cls:"IfcCartesianPoint"}),
(roofslab2{cls:"IfcSlab"})-[]-
({cls:"IfcProductDefinitionShape"})
-[]-({cls:"IfcShapeRepresentation"})-[]-
({cls:"IfcExtrudedAreaSolid"})-[]-
(pro2{cls:"IfcRectangleProfileDef"})
```

Code Listing 5.9: Matching original reference Point and Rectangle Profile Definition of two arbitrary Roof Slabs by filtering with Patterns

With these premise the coordinates of the four corners of rectangle roof slabs can be derived from the lower left corner point, the width and the length. As shown in the graph, taken that the original point is (x, y), length is XDim and width is YDim, the other three nodes are $(x + XDim, y)$, $(x + XDim, y + YDim)$ and $(x, y + YDim)$.

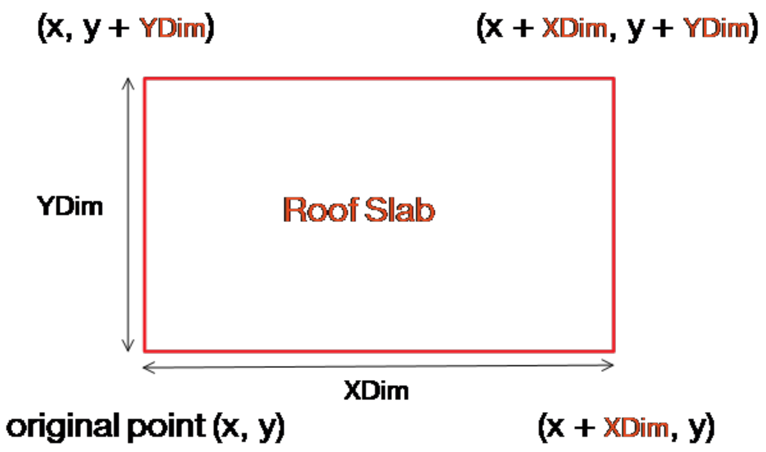


Figure 5.19: Deriving Coordinates of the four Corners of rectangle Roof Slab

The exact procedure to calculate the coordinates in Neo4j uses the same method mentioned above. The coordinates of the first slab are defined as x_{i1}, y_{i1}, in which i varies from 1 to 4. Accordingly the positions of second slab are (x_{i2}, y_{i2}). The roof slab names are also indexed as slab to enable combination of results, since the *UNION* function required same parameter to be returned.

```
WITH
origin1.CoordinateX AS x11,origin1.CoordinateY AS y11,
origin1.CoordinateX + pro1.XDim AS x21,
origin1.CoordinateY AS y21,
origin1.CoordinateX + pro1.XDim AS x31,
origin1.CoordinateY + pro1.YDim AS y31,
origin1.CoordinateX AS x41,
origin1.CoordinateY + pro1.YDim AS y41,

origin2.CoordinateX AS x12, origin2.CoordinateY AS y12,
origin2.CoordinateX + pro2.XDim AS x22,
origin2.CoordinateY AS y22,
origin2.CoordinateX + pro2.XDim AS x32,
origin2.CoordinateY + pro2.YDim AS y32,
origin2.CoordinateX AS x42,
origin2.CoordinateY + pro2.YDim AS y42,
roofslab1.Name AS slab1, roofslab2.Name AS slab2,
storey.Name AS storey
```

Code Listing 5.10: Calculation of Corner Node Coordinates of Roof Slabs and index Roof Slab Names and Storeys

In contrast the position of floor slab is deduced by its geometric description in *IfcExtrudedAreaSolid* and set in the centre of the rectangle, which meets the default requirement of *IfcRectangleProfileDef* and need no offset vector.

Finding original reference point and rectangle profile definition of two arbitrary floor slabs.

```
MATCH
(floorslab1{cls:"IfcSlab"}),(origin1{cls:"IfcCartesianPoint"}),
(pro1{cls:"IfcRectangleProfileDef"}),
(storey{cls:"IfcBuildingStorey"}),(floorslab2{cls:"IfcSlab"}),
(origin2{cls:"IfcCartesianPoint"}),
(pro2{cls:"IfcRectangleProfileDef"})
WHERE
(storey)<-[]-({cls:"IfcRelContainedInSpatialStructure"})
-[]->(floorslab1)
AND
(floorslab1)-[]-({cls:"IfcProductDefinitionShape"})-[]-
({cls:"IfcShapeRepresentation"})-[]-
({cls:"IfcExtrudedArea-Solid"})
-[]-({cls:"IfcAxis2Placement3D"})-[]-(origin1)
AND
(floorslab1)-[]-({cls:"IfcProductDefinitionShape"})-[]-
({cls:"IfcShapeRepresentation"})-[]-
({cls:"IfcExtrudedAreaSolid"})-[]-(pro1)
AND
(storey)<-[]-({cls:"IfcRelContainedInSpatialStructure"})
-[]->(floorslab2)
AND
(floorslab2)-[]-({cls:"IfcProductDefinitionShape"})-[]-
({cls:"IfcShapeRepresentation"})-[]-
({cls:"IfcExtrudedAreaSolid"})
-[]-({cls:"IfcAxis2Placement3D"})-[]-(origin2)
AND
(floorslab2)-[]-({cls:"IfcProductDefinitionShape"})-[]-
({cls:"IfcShapeRepresentation"})-[]-
({cls:"IfcExtrudedAreaSolid"})-[]-(pro2)
```

Code Listing 5.11: Matching original reference Point and Rectangle Profile Definition of two arbitrary Floor Slabs by filtering with Patterns

Firstly, the needed storey, slab, placement and profile classes are matched for further reference. Then they are filtered and organized by their mutual relationships, through the specified patterns in which the desired nodes are. Two floor slabs need to link to same storey class to ensure that they are at the same elevation. The rectangle profile and cartesian points should belong to the sample slab, instead of to other irrelevant entities.

The coordinates of the four corners of rectangle floor slabs can be derived from the center point, the width and the length. As shown in the graph, taken that the original point is (x, y), length is $XDim$ and width is $YDim$, the four corner nodes are $(x - Xdim/2, y - Ydim/2)$, $(x + Xdim/2, y - Ydim/2)$, $(x + Xdim/2, y + Ydim/2)$ and $(x - Xdim/2, y + Ydim/2)$.

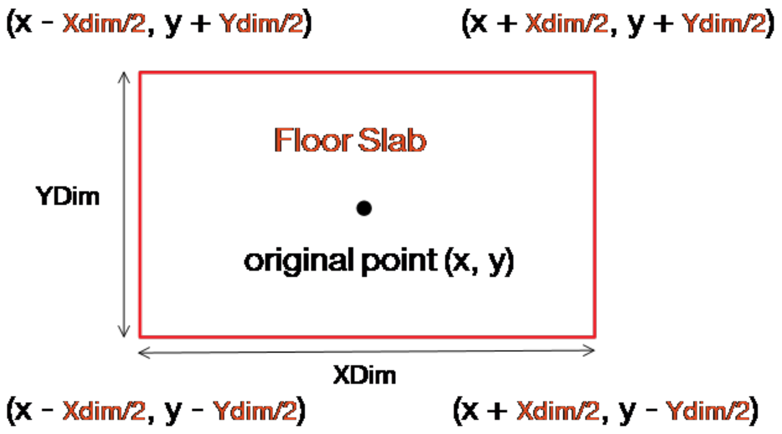


Figure 5.20: Deriving Coordinates of the four Corners of rectangle Floor Slab

The difference from the previous calculation lies in the calculation of coordinates. Here are the positions derived as mentioned above: $(x-Xdim/2, y-Ydim/2)$, $(x+Xdim/2, y-Ydim/2)$, $(x+Xdim/2, y+Ydim/2)$ and $(x-Xdim/2, y+Ydim/2)$. Similarly are floor slabs indexed as slabs.

```
WITH
origin1.CoordinateX - pro1.XDim/2 AS x11,
origin1.CoordinateY - pro1.YDim/2 AS y11,
origin1.CoordinateX + pro1.XDim/2 AS x21,
origin1.CoordinateY - pro1.YDim/2 AS y21,
origin1.CoordinateX + pro1.XDim/2 AS x31,
origin1.CoordinateY + pro1.YDim/2 AS y31,
origin1.CoordinateX - pro1.XDim/2 AS x41,
origin1.CoordinateY + pro1.YDim/2 AS y41,
origin2.CoordinateX - pro2.XDim/2 AS x12,
origin2.CoordinateY - pro2.YDim/2 AS y12,
origin2.CoordinateX + pro2.XDim/2 AS x22,
origin2.CoordinateY - pro2.YDim/2 AS y22,
origin2.CoordinateY - pro2.YDim/2 AS y22,
origin2.CoordinateY + pro2.YDim/2 AS y32,
origin2.CoordinateX - pro2.XDim/2 AS x42,
origin2.CoordinateY + pro2.YDim/2 AS y42,
floorslab1.Name AS slab1, floorslab2.Name AS slab2,
storey.Name AS storey
```

Code Listing 5.12: Calculation of Corner Node Coordinates of Floor Slabs and index Roof Slab Names and Storeys

5.3.3.2 Slab Overlap Dection

After knowing the coordinates of the corner node, we need to check if these two rectangles, composed of eight nodes in Neo4j, overlap. In the line intersection check mentioned above in Section 6.2.1.2, it is indicated that the first step of line segment intersection detection is bounding box overlap check. As mentioned, bounding boxes are boxes around line segments such that endpoints of both line segments are at the corner of the boxes and edges of the boxes are in parallel to the coordinate axes. One bounding box can be regarded as one slab since they have a rectangular shape. Obviously by replacing the two ends of the line segment with the diagonal corners of the rectangle, bounding box check can be equivalently applied.

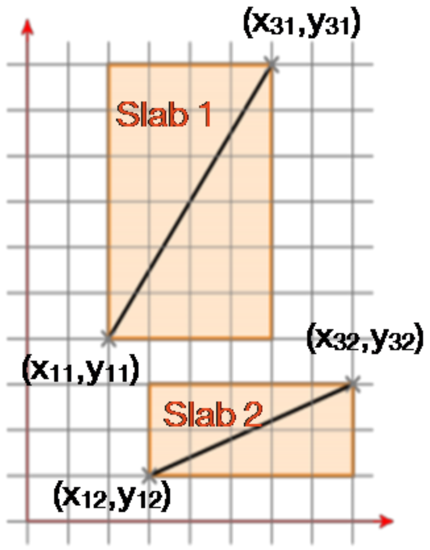


Figure 5.21: Bounding Box formed by two diagonal Nodes of the Slab

Here, the lower left corner (x_1, y_1) and upper right corner (x_3, y_3) of the rectangle are set as two ends of the line segment, because the bounding box that they form can fully represent the shape of the roof.

The basic algorithm of the program is:

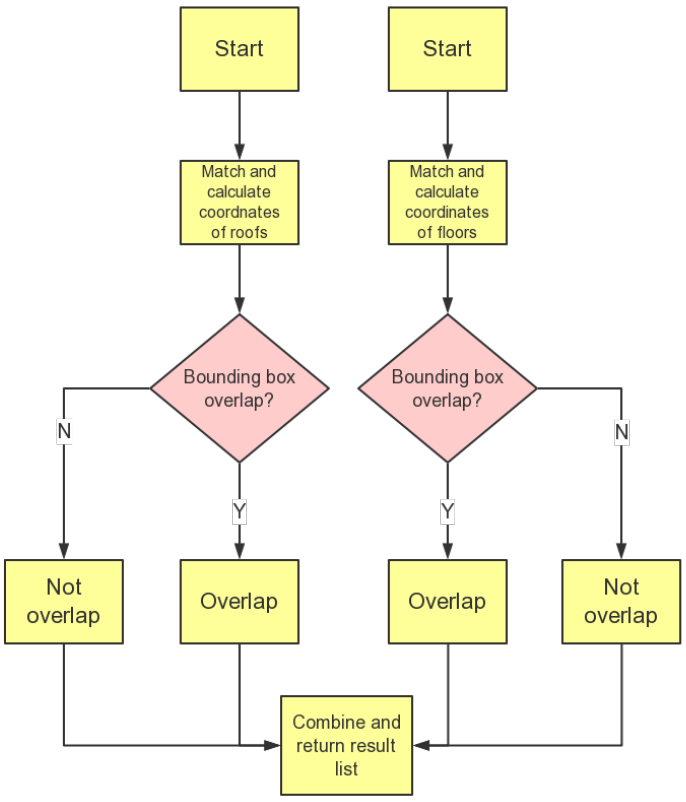

Start
Start
Match and calculate coordnates of roofs
Match and calculate coordinates of floors
Bounding box overlap?
Bounding box overlap?
N
Y
Y
N
Not overlap
Overlap
Overlap
Not overlap
Combine and return result list

Figure 5.22: Algorithm of deciding whether two slabs overlap

The algorithm of the syntax is:

1. Match and find all the building storey, original points and rectangle profiles of floor slabs and roof slabs separately.
2. Calculate the coordinates of the corner nodes based on original position, length and width.
3. Apply bounding box check on slabs in the same storey. Return "Overlap" if they overlap and "Not Overlap" if they don't touch each other. Return "Same Slab" if there is only one floor slab in the corresponding storey. Also, return the corresponding slab name.
4. Combine the result of floor slab check and roof slab check using *UNION* function in Cypher, return a list.

```
MATCH
......
UNWIND [x11,x31] AS line1x UNWIND [x12,x32] AS line2x
UNWIND [y11,y31] AS line1y UNWIND [y12,y32] AS line2y
```

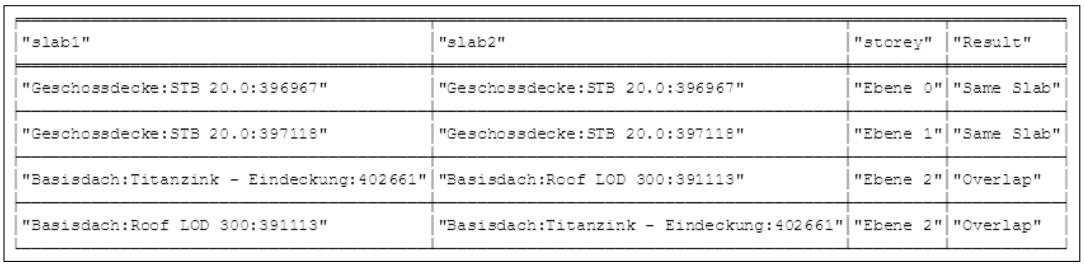
```
RETURN slab1 AS slab1, slab2 AS slab2, storey AS storey,
CASE WHEN max(line1x)<min(line2x) OR max(line2x)<min(line1x)
OR max(line1y)<min(line2y) OR max(line2y)<min(line1y)
THEN "Not Overlap" WHEN slab1 = slab2 THEN "Same Slab"
ELSE "Overlap" END AS Result

UNION

MATCH
......
UNWIND [x11,x31] AS line1x UNWIND [x12,x32] AS line2x
UNWIND [y11,y31] AS line1y UNWIND [y12,y32] AS line2y
RETURN slab1 AS slab1, slab2 AS slab2, storey AS storey,
CASE WHEN max(line1x)<min(line2x) OR max(line2x)<min(line1x)
OR max(line1y)<min(line2y) OR max(line2y)<min(line1y)
THEN "Not Overlap" ELSE "Overlap" END AS Result
```

Code Listing 5.13: Decision Statement which consists of Bounding Box Check to detect whether two Slabs overlap, the Result of Floor Slabs and Roof Slabs are combined

Similar to previous boundary box check in 6.2.1.2, it is not possible for the lines to meet when:

1. The maximal x value of line A is smaller than minimal x value of line B
2. The maximal x value of line B is smaller than minimal x value of line A
3. The maximal y value of line A is smaller than minimal y value of line B
4. The maximal y value of line B is smaller than minimal y value of line A

If any of these conditions are met, then return "Not Overlap". When floor slab that are in the same storey have the same names, then return "Same Slab". In other cases display "Overlap" to warn the clash between them. In all of the cases will the slab names and storeys be exported. At the end combine the query result of the respective detection between floors and roofs.



"slab1"	"slab2"	"storey"	"Result"
"Geschossdecke:STB 20.0:396967"	"Geschossdecke:STB 20.0:396967"	"Ebene 0"	"Same Slab"
"Geschossdecke:STB 20.0:397118"	"Geschossdecke:STB 20.0:397118"	"Ebene 1"	"Same Slab"
"Basisdach:Titanzink - Eindeckung:402661"	"Basisdach:Roof LOD 300:391113"	"Ebene 2"	"Overlap"
"Basisdach:Roof LOD 300:391113"	"Basisdach:Titanzink - Eindeckung:402661"	"Ebene 2"	"Overlap"

Figure 5.23: Result floor Slab and Roof Slab Overlap Check

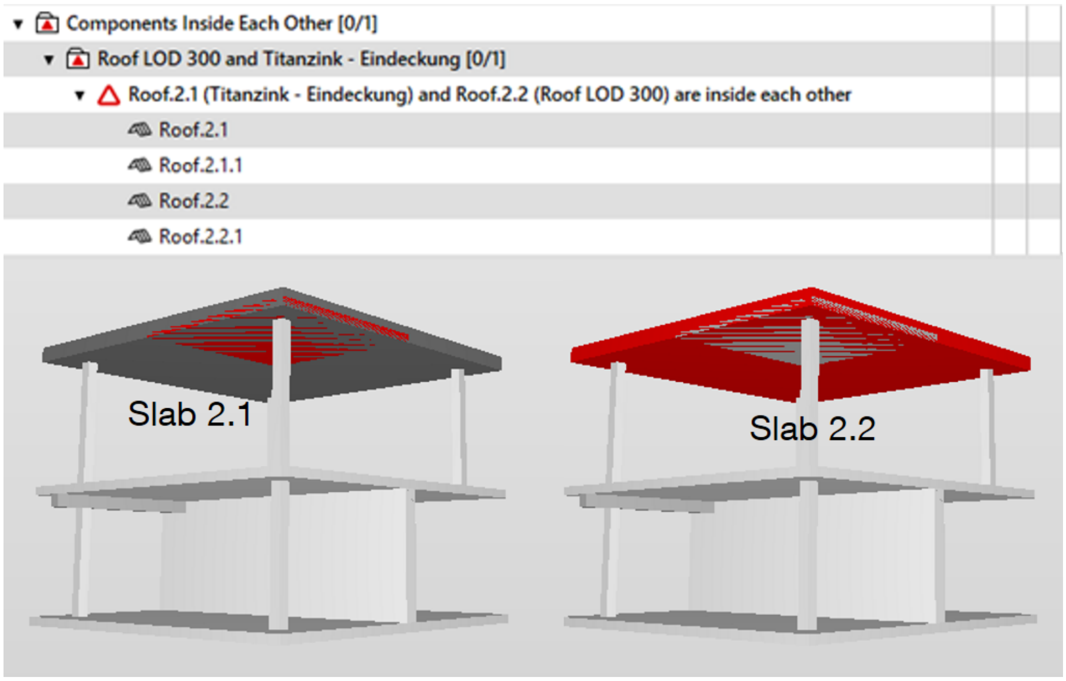

▾ Components Inside Each Other [0/1]
▾ Roof LOD 300 and Titanzink - Eindeckung [0/1]
▾ Roof.2.1 (Titanzink - Eindeckung) and Roof.2.2 (Roof LOD 300) are inside each other
Roof.2.1
Roof.2.1.1
Roof.2.2
Roof.2.2.1

Figure 5.24: Roof Intersection Check Result in Solibri and Position of overlapped Roofs

The result shows that there is only one slab in the ground floor and first floor, which makes it not possible to clash with another slab in the same storey. The third and fourth line indicate same overlap between slab with material Titanzik and slab defined as Roof LOD 300. They may partially or totally overlap and need to be fixed using other modelling software. The result is the same as what the Solibri roof intersection check provides. Their positions are marked in the Figure 5.24. The advantage of using graph data is demonstrated in this example. Through precise definition of paths can diverse demanded data, like origins and profiles of floor slabs and roof slabs which are represented against different rules, be retrieved specifically. Moreover, additional information such as storeys and materials that are linked to the queried entities can be exported together with the result. This feature is useful when the overlap detection result is purposed to fire protection inspection or quantity take-off, since additional helpful information can be retrieved at the same time.

Further applications, such as returning overlapped area, enabling the check of slabs in special shape with variable thicknesses and displaying check result back in a 3D-model, can be developed with the help of a better geometry interpreter and user interface.

Chapter 6

Conclusion

Filtering and querying properties in different LODs and performing model checks on building elements have demonstrated the capability of graph databases to explore and analyse BIM models. The objectified relationships in IFC are sufficiently represented by directional edges. Classes describing different entities in IFC are modelled as nodes and their properties. Along with the abundant semantic content in IFC, various queries and extractions can be performed in graph databases. Based on the demonstrations presented in the previous chapters, the following can be concluded:

1. IFC data can be interpreted with an external parser or library into Cypher language, so as to import it into graph databases. The interpreters can be individually edited or improved to export desired information and enhance connectivity.

2. The Neo4j database is able to retrieve and return information in a list like a common relational database. Queries can be not only for basic properties like width and length, but also for individual requirements like thermal performance or number of openings. Based on the examples in Chapter 5, users can easily develop their own query and generate customized data lists. Furthermore, Neo4j can also display properties graphically with nodes and patterns to explain the relationships or developments between them more clearly.

3. Models at different LODs can be loaded into the same graph to trace the development and variations by applying data retrieval on all the available LODs. The required property sets in different LODs can be customized according to specific demands of different design teams. Data nodes can be coloured, labelled and connected correspondingly. Accordingly, the different LOD models' reliability can be explored.

4. IFC data includes geometry representations, which allows BIM-based model checks, such as clash and deficiency detection based on key shape features, like axis and position

points, to be executed in the graph database. The results can be displayed not only with lists but also graphically. The answers for realistic problems like intersection detection are in accordance with commercial model checking software Solibri.

5. Graph databases have high accuracy in retrieving specific information by filtering or navigating with their labels, properties, relationships and patterns. Therefore even if this information is expressed differently in IFC, the correct item can be found. Other relevant information besides initial objectives can also be searched for and returned in one query. Furthermore, this advantage provides numerous application possibilities when designing queries for specific needs such as searching for fire walls in special orientations.

However, from the above demonstrations, several limitations have been identified in this study and may affect the applicability of graph database in querying BIM models. The underlying causes may relate to the representation methods of IFC data and the processing focus of Neo4j.

1. The quality and content of imported data in graph databases is dependent on IFC interpreters, which may utilize another BIM server or be written in other programming languages like Python or Ruby. Editing the interpreter requires adequate programming knowledge as well as an understanding of relevant BIM servers. This might be complicated for general users.

2. Neo4j does not support many spatial and mathematical functions. This makes it complicated for writing queries regarding geometry operations. Its conditional statements are also limited to returning literal expressions. However, this limitation can be compensated by applying external geometry parsers and function plug-ins.

3. Solid models in IFC can have different shape representations. When an objects is represented by faceted boundary representations, which is common when the shape is more complicated, retrieving its basic features could be unpractical compared to shapes described by swept solid or Boolean result. This restricts the shapes of objects to simple and regular ones. This can also be solved by introducing external geometry interpreters and passing the result back to Neo4j.

Future work could involve designing a more user-friendly and automatic interface to allow more convenient and feasible BIM model analysis. More importantly, utilizing external libraries about spatial operations and additional functions could significantly expand the capacity and scope of model checks in graph databases. These could include deficiency check for the missing structural components along the load-bearing path or the fire compartment design. Then, the returned results could not only be displayed in Neo4j, but also projected in the 3D model to reposition them quickly.

In conclusion, the methodology proposed in this study is an initial step in the direction of utilizing graph databases to analyse IFC-based BIM models. Although there are currently limitations in this mechanism, graphs can be considered a potential and promising tool in model checking and LOD evaluation.

Appendix A

Appendix

1. The ifc2neo4j.py converter used in the study to transfer IFC data into Cypher language. It was created by user ysangkok (https://gist.github.com/ysangkok/8aa7ab1c3207536518f3c3bf5f-664880) and edited by me.

```
import re
import sys
import os.path
import ifcopenshell
import itertools
import json

def chunks2(iterable,size,filler=None):
it = itertools.chain(iterable,itertools.repeat(filler,size-1))
chunk = tuple(itertools.islice(it,size))
while len(chunk) == size:
yield chunk
chunk = tuple(itertools.islice(it,size))

class IfcTypeDict(dict):
def __missing__(self, key):
value = self[key] =
    ifcopenshell.create_entity(key).wrapped_data.get_attribute_names()
return value

typeDict = IfcTypeDict()

assert typeDict["IfcWall"] == ('GlobalId', 'OwnerHistory', 'Name',
    'Description', 'ObjectType', 'ObjectPlacement', 'Representation', 'Tag')

nodes = []
edges = []
#wallid = None
```

```
ourLabel = sys.argv[2]

f = ifcopenshell.open(sys.argv[1])
for el in f:
tid = el.id()
cls = el.is_a()
pairs = []
keys = []
try:
keys = [x for x in el.get_info() if x not in ["type", "id"]]
except RuntimeError:
# we actually can't catch this, but try anyway
pass
for key in keys:
val = el[key]
if any(hasattr(val,"is_a") and val.is_a(thisTyp) for thisTyp in [
    "IfcBoolean", "IfcLabel", "IfcText", "IfcReal"]):
val = val.wrappedValue
if type(val) not in (str, bool, float):
continue
pairs.append((key, val))

nodes.append((tid, cls, pairs))
for i in range(len(el)):
try:
el[i]
except RuntimeError as e:
if str(e) != "Entity not found":
print("ID", tid, e, file=sys.stderr)
continue
if isinstance(el[i], ifcopenshell.entity_instance):
if el[i].id() != 0:
edges.append((tid, el[i].id(), typeDict[cls][i]))
continue
else:
print("attribute " + typeDict[cls][i] + " of " + str(tid) + " is zero",
    file=sys.stderr)
try:
iter(el[i])
except TypeError:
continue
destinations = [x.id() for x in el[i] if isinstance(x,
    ifcopenshell.entity_instance)]
for connectedTo in destinations:
edges.append((tid, connectedTo, typeDict[cls][i]))
if len(nodes) == 0:
print("no nodes in file", file=sys.stderr)
sys.exit(1)
```

```
indexes = set(["nid", "cls"])

for chunk in chunks2(nodes, 100):
idx = 0
print("CREATE ", end="")
for i in chunk:
if i is None: continue
nId, cls, pairs = i
if idx != 0: print(",",end="")
idx = idx + 1

pairsStr = ""
for k,v in pairs:
indexes.add(k)
pairsStr += ", " + k + ": " + json.dumps(v)

print("(a" + str(idx) + ":" + ourLabel + " { nid: " + str(nId) + ",cls: '" +
    cls + "'" + pairsStr + " })", end="")
print(";")

for idxName in indexes:
print("CREATE INDEX on :" + ourLabel + "(" + idxName + ");")

#print("CREATE ")
#ind = 0
#for (nId1, nId2, relType) in edges:
#print("(a" + repr(ind) + ":" + ourLabel + " { nid: " + nId1 + " })" + "-[r"
    + repr(ind) +":"+ relType + " ]->(b" + repr(ind) + ":" + ourLabel + " {
    nid: " + nId2 + " }),")
#print("(a" + str(ind) + ":" + ourLabel + " { nid: " + nId1 + " })" + "-[r"
    + str(ind) + ":" + relType + " ]->(b" + str(ind) + ":" + ourLabel + " {
    nid: " + nId2 + " }),")
#print("(a",ind,":",ourLabel,"{
    nid:",nId1,"})","-[r",ind,":",relType,"]->(b",ind,":",ourLabel,"{nid:",
    nId2,"}),", sep='')
#print("(a{}:{}{nid:{}})-[r{}:{}]->(b{}:{}{nid:{}}),
    ".format(ind,ourLabel,nId1,ind,relType,ind,ourLabel,nId2))
#print("(a",nId1,":",ourLabel,"{
    nid:",nId1,"})","-[:",relType,"]->(b",nId2,":",ourLabel,"{nid:",nId2,"}),",
    sep='')
#print("MATCH (a",nId1,":",ourLabel,"),(b",nId2,":",ourLabel,") WHERE
    a",nId1,".nid = ",nId1," AND b",nId2,".nid = ",nId2," CREATE
    (a",nId1,")-[:",relType,"]->(b",nId2,")",sep='')
#ind = ind + 1

for (nId1, nId2, relType) in edges:
print("MATCH (a", nId1, ":", ourLabel, "),(b", nId2, ":", ourLabel, ") WHERE
    a", nId1, ".nid = ", nId1, " AND b",nId2, ".nid = ", nId2, sep='')
for (nId1, nId2, relType) in edges:
```

```
print("CREATE (a", nId1, ")-[:", relType, "]->(b", nId2, ")", sep='')

#print("MATCH a=(first:IfcNode {nid: " + str(wallid) +
    "})-[:RELTYPE*1..2]-(other {cls: \"IFCWINDOW\"}) RETURN distinct other;")
```

Bibliography

Autodesk, I. (2018). Autodesk Revit IFC manual.

Batra, S. & Tyagi, C. (2012). Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)* 2(2), S. 509–512.

BIMForum (2017). Level of Development Specification Guide.

Bloomberg, M. R., Burney, D. & Resnick, D. (2012). BIM guidelines. *New York City Department of Design and Construction*, S. 1–57.

Bolpagni, M. (2016). The many faces of 'LOD', BIM Think Space.

Borrmann, A. (2016). *Building Information Modeling Data exchange and interoperability.* TUM, Chair of Computational Modeling and Simulation.

Hjelseth, E. (2015). BIM-based model checking (BMC). *Building Information Modeling–Applications and Practices*, S. 33–61.

Holness, G. (2006). Building information modeling. *ASHRAE journal* 48(8), S. 38–46.

Hörtnagl, E. (2017). LODs – Der Fertigstellungsgrad.

IfcOpenShell (2018). http://ifcopenshell.org/. [Online; accessed 22-September-2018].

Ismail, A., Nahar, A. & Scherer, R. (2017). Application of graph databases and graph theory concepts for advanced analysing of BIM models based on IFC standard. *Proceedings of EGICE*.

Khemlani, L. (2004). Autodesk Revit: implementation in practice. *White paper, Autodesk*.

Liebich, T. (2009). IFC 2x Edition 3. In: *Model implementation guide. version 2.0.* AEC3 Ltd.

Miller, J. J. (2013). Graph database applications and concepts with Neo4j. In: *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, Volume 2324, S. 36.

Nationalbimlibrary.com (2018). NBS National BIM Library. https://www.nationalbimlibrary.com/en/. Accessed June, 2018.

Rijksgebouwendienst, R. B. (2012). Standard. *Rijksgebouwendienst, Netherlands* 803.

Sarwar, B., Karypis, G., Konstan, J. & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In: *Proceedings of the 10th international conference on World Wide Web*, S. 285–295. ACM.

Solibri (2014). Getting Started with Solibri Model Checker^{TM}.

Statsbygg (2017). Statsbygg Building Information Modelling Manual Version 1.2. 1 (SBM1. 2.1). *Retrieved from.*

Thoma, M. (2013). How to check if two line segments intersect. https://martin-thoma.com/how-to-check-if-two-line-segments-intersect/. Accessed October, 2018.

Treldal, N., Vestergaard, F. & Karlshøj, J. (2016). Pragmatic Use of LOD–a Modular Approach.

Van Berlo, L. & Bomhof, F. (2014). Creating the Dutch national BIM levels of development. In: *Computing in Civil and Building Engineering (2014)*, S. 129–136.

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y. & Wilkins, D. (2010). A comparison of a graph database and a relational database: a data provenance perspective. In: *Proceedings of the 48th annual Southeast regional conference*, S. 42. ACM.

Wikipedia contributors (2018a). Neo4j — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Neo4j&oldid=866850985. [Online; accessed 14-November-2018].

Wikipedia contributors (2018b). PyCharm — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=PyCharm&oldid=854078647. [Online; accessed 22-September-2018].

Winter, S. (2016). *Baukoskript 2400 Flachdach.* TUM, Chair of Timber Structures and Building Construction.

Declaration of Originality

With this statement I declare, that I have independently completed this Master's thesis. The thoughts taken directly or indirectly from external sources are properly marked as such. This thesis was not previously submitted to another academic institution and has also not yet been published.

München, 19. November 2018

Sining Xu

Sining Xu
Adelheidstr. 15
D-80798 München
e-Mail: sining.xu@tum.de