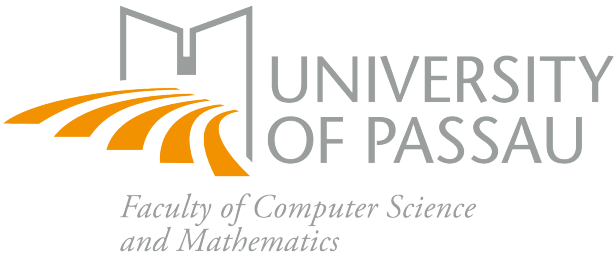



# Dissertation

submitted to the

Faculty of Computer Science and Mathematics
University of Passau

# Towards Storing 3D Model Graphs in Relational Databases

Matthias Schmid

July 21, 2021

*Supervisor:* Prof. Dr. Burkhard Freitag

A dissertation submitted to the faculty of computer science and mathematics in partial fulfillment of the requirements for the degree of doctor of engineering sciences.

*1st reviewer:* Prof. Dr. Burkhard Freitag
*2nd reviewer:* Prof. Dr. Alfons Kemper

# Abstract

The increasing relevance of massive graph data reinforces the need for adequate graph data management. While several graph database engines have been developed, the storage of graph data in a relational database management system, and therefore the seamless integration into existing information systems remains an open challenge.

Motivated by the use case to integrate *Building Information Modeling (BIM)* data into the *MonArch* system, we propose a solution that transforms the *BIM* data into a property graph and stores this graph in the database system.

We present a novel approach to efficiently store property graph data in a relational database management system using *JSON* functionality and redundant storage of edges in adjacency lists and show how to import huge data sets into this schema. Applying this approach, we import data sets of up to nearly 1 TB of disk space within the relational database, while only having 96 GB of main memory available.

We also present a new approach of how to retrieve data from this database schema, translating queries written in the popular property graph query language *Cypher* into *SQL*. Hence, we provide an intuitive way to write semantically complex queries.

We also demonstrate the efficiency of our approach using the standardized *Linked Data Benchmark Council – Social Network Benchmark (LDBC - SNB)* framework. Our approach increases the throughput for this benchmark by up to 85 times, compared to existing approaches for RDBMS.

In addition, we propose a new method to transform BIM data into the property graph model and how to apply the aforementioned property graph storage to this data. We can import *IFC* models of up to 300 MB within five minutes.

We show the suitability of our approach using our own use case specific benchmark, which we integrated into the previously mentioned *Social Network Benchmark*. For our interactive use case-specific queries, we achieve response times faster than 5 ms in 99% of all executions.

Finally, we present how the aforementioned approach to store BIM data in a relational database management system is integrated into the existing *MonArch* system by splitting the different functionalities of our approach into a microservice architecture.

# Kurzfassung

Die steigende Relevanz von riesigen Graphdatenmengen verstärkt die Notwendigkeit von adäquatem Graphdaten Management. Während bereits mehrere Graphdatenbanken entwickelt wurden, bleibt die Speicherung von Graphdaten in relationalen Datenbanken und die damit verbundene nahtlose Integration in bereits existierende Informationssysteme eine ungelöste Herausforderung.

Motiviert durch unseren eigenen Anwendungsfall *Building Information Modeling (BIM)* Daten in das *MonArch* Informationssystem zu integrieren, schlagen wir einen Ansatz vor, *BIM* Daten in eine Property Graph Form umzuwandeln und diesen in der Datenbank zu speichern.

Um dies zu erreichen, stellen wir einen neuartigen Ansatz vor, um Property Graphen in einem relationalen Datenbanksystem zu speichern, indem wir Funktionalitäten wie *JSON* und die redundante Speicherung von Kanten in Adjazenzlisten kombinieren und zeigen wie große Mengen dieser Daten in das Schema importiert werden können. Durch die Anwendung unseres Ansatzes können wir Datensätze von bis zu 1 TB in das Datenbanksystem importieren, während wir nur 96 GB Hauptspeicher zur Verfügung haben.

Wir stellen außerdem einen neuen Ansatz vor, um Daten aus dem zuvor genannten Schema abzufragen, indem wir die beliebte Graphanfragesprache *Cypher* in die Sprache *SQL* übersetzen. Dadurch erreichen wir eine intuitive Art semantisch komplexe Anfragen zu schreiben.

Zusätzlich zeigen wir die Effizienz unseres Ansatzes, indem wir das standardisierte Evaluationsframework *Social Network Benchmark* des *Linked Data Benchmar Council (LDBC – SNB)* verwenden. Unser Ansatz erhöht den Durchsatz dieses Benchmarks, im Vergleich zu existierenden Ansätzen für relationale Datenbanksysteme, auf einen bis zu 85-fachen Durchsatz.

Zusätzlich schlagen wir eine neue Methode vor, um *BIM* Daten in das Property Graph Modell zu übertragen und wie das zuvor vorgestellte Speichermodel verwendet werden kann, um diese Daten zu speichern. Damit können wir *IFC* Modelle mit bis zu 300 MB in unter 5 Minuten in unser System importieren.

Schließlich zeigen wir die Eignung unseres Ansatzes, indem wir einen eigenen Benchmark spezifisch für unseren Anwendungsfall verwenden, welchen wir in den zuvor erwähnte *Social Network Benchmark* integriert haben. Für unsere anwendungsfallspezifischen Anfragen erreichen wir Antwortzeiten von unter 5 ms in 99% der Ausführungen.

Zu guter Letzt präsentieren wir, wie der zuvor vorgestellte Ansatz zur Speicherung von *BIM* Daten in einem relationalen Datenbankmanagementsystem in das existierende *MonArch* System integriert werden kann, indem wir die verschiedenen Funktionalitäten unseres Ansatzes in Microservices aufteilen.

# Contents

# Part I.

# Preface

# 1. Introduction

The increasing relevance of massive graph data reinforces the need for adequate graph data management. To meet this requirement of property graph storage, several graph databases engines have been developed. Some of the more popular[1] examples of databases with graph storage capability include *Neo4j*[2], *Microsoft Azure Cosmos DB*[3], and *OrientDB*[4].

While most of the existing work targets the support of the graph data model in native stores or in key-value stores, only few make use of the well-researched relational systems. Relational databases offer a number of significant advantages, because they have full ACID compliance, concurrency support and sophisticated query optimization while performing well — mostly independent of data size. Most native graph solutions offer good performance as long as the data fits into main memory. However, due to the fast growth of graph data we cannot assume that we will be able to fully load graphs into main memory. This leads to another significant advantage of relational systems. The seamless integration of graph data with classical relational data is not well researched and remains an open challenge – especially when graph data needs to be integrated with already existing information systems.

## 1.1. Motivation

The research for this thesis was originally triggered by a use case in which we need to store building data, Resource Description Framework (RDF) data that describes additional information about those buildings and relational data that originates from an existing application into a single information system: The *MonArch System* [FS17].

*MonArch* is an information system designed for documenting structures such as architectural objects, urban situations, and archaeological sites. A screenshot of the *MonArch* user interface is depicted in Figure 1.1. A *MonArch* database contains a digital model of the structure, i.e. a digital representation of the building, ensemble or site.

The trend to use 3D models as a source of information in the field of building construction and maintenance through the method of *Building Information Modeling (BIM)* can be observed for several years now. The central piece of the *BIM* process is a 3D model of the building.

[1] from https://db-engines.com/de/ranking/graph+dbms, if not explicitly stated, all websites were accessed in July 2021

[2] https://www.neo4j.com/

[3] https://azure.microsoft.com/de-de/services/cosmos-db/

[4] https://orientdb.org

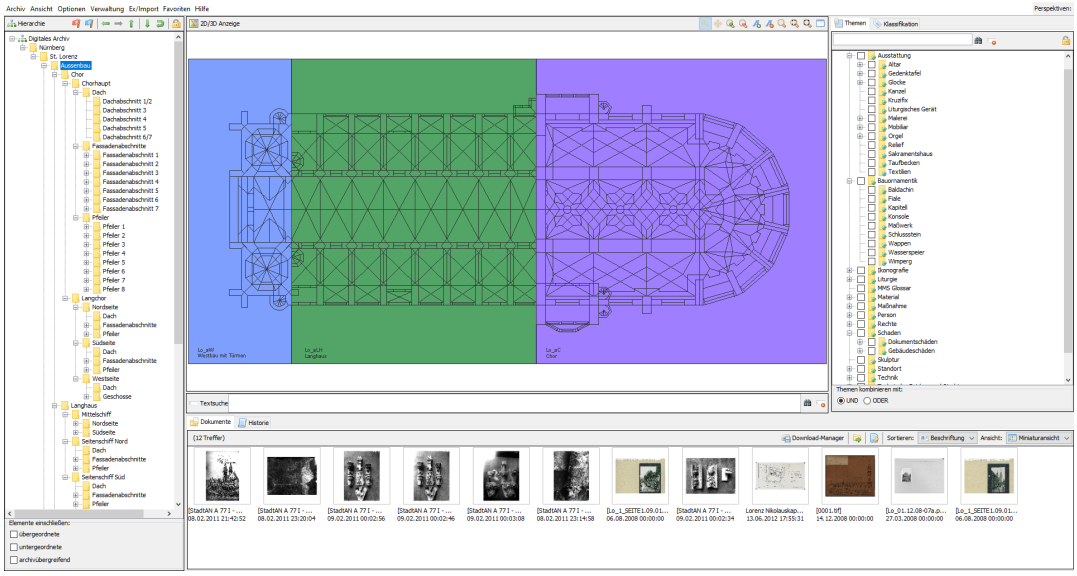


Figure 1.1.: The original user interface of *MonArch*.

In contrast to other 3D model approaches, a model originating from *BIM* already offers a segmentation in components of the building, which is required to link additional information to the building components. The model also provides complex spatial and logical relationships between those components. The standardized *Industry Foundation Classes (IFC)* offers an open exchange format for this type of 3D building data, which makes the data accessible for any application.

As depicted in Figure 1.2, our goal was to integrate 3-dimensional building data into the *MonArch* information system. The building model is created in specialized Computer Aided Design (CAD) software tools. Using an appropriate exchange format, the building should be imported into the *MonArch* building store. Here it can be linked with additional information like arbitrary digital documents, topic and semantic information. In addition, the *MonArch* system should be able to retrieve information about the building like the building structure and properties of components. These can then be displayed to the user and individual values can also be altered.

Using an export mechanism, the same building can then be exported and be edited again in the CAD software. After repeating the building model import, the previously created persistent links should still be viable. The *MonArch* system is implemented using the **PostgreSQL** Relational Database Management System (RDBMS) and therefore one of our goals was to integrate the building data directly into a RDBMS. A more detailed description of our use case can be found in Chapter 7.

While a standardized exchange format exists as the *Industry Foundation Classes (IFC)* data model, there are only few database-based solutions to store and query *IFC* data. To the best of our knowledge, none of those solutions offer support to query arbitrary data.

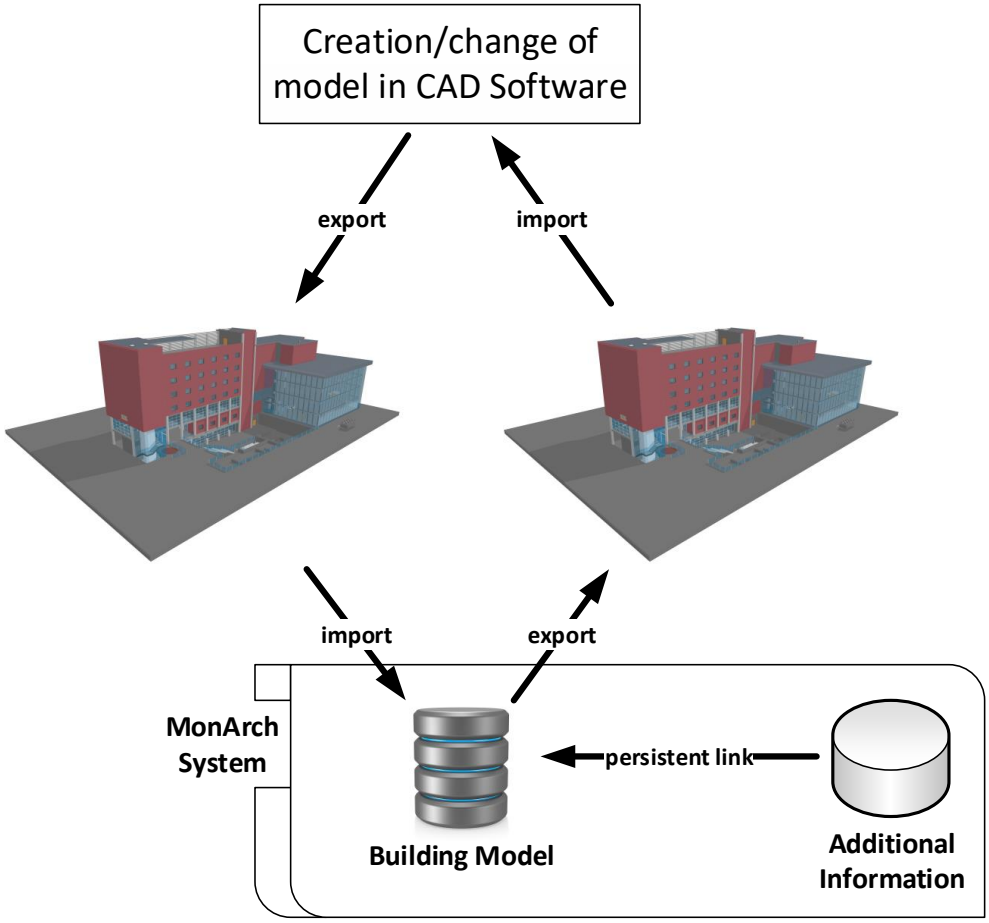



Figure 1.2.: The desired import/export cycle for building models in the *MonArch* system.

During our initial research we found that storing the IFC data as a property graph is an elegant and efficient method to store the building data. Subsequently, we needed to find a way to store a property graph efficiently in a Relational Database Management System (RDBMS). Since writing queries for this storage approach can get tedious, we also propose an efficient method to write queries for our storage approach using the *Cypher* query language.

## 1.2. Contributions

The contributions of this work can be split into two parts:

The first part of our work targets the storage of property graph data in standard state-of-the-art Relational Database Management System. While existing work achieves good performance [Sun+15], it can be improved provided certain assumptions hold. In particular, our approach significantly improves the performance for workloads that mainly require read-only queries. For the remainder of this work, we call our approach **Relational Adjacency Table Graph (RATG)**. Notably, we achieve the following goals:

1. We present a **novel schema for storing arbitrary property graphs** in any desired state-of-the-art relational database management systems. The schema uses the widespread JavaScript Object Notation (JSON)-support of current database systems. No other particular requirements for the relational database have to be met. By shredding different edge types into the columns of adjacency tables using a hashing function we minimize the number of tuples that have to be fetched to query all neighbors of a given node.
2. We show how the required **hash functions** can be **computed from** underlying **data models**. The approach offers two outputs: First, the number of required columns needed to store all edges of any vertex in a single row and second, the associated hash function.
3. We present an **efficient data import** for huge graph data sets. Since our schema requires the redundant storage of edges in three different tables (edge attributes, outgoing and incoming adjacency) and the data normally does not fit in main memory, the efficient import of data is a challenge in itself.
4. The schema is evaluated based on the **LDBC-SNB Interactive Workload**, a standardized property graph benchmark. First results show:
    - Our approach offers **excellent performance** on **read-only queries**.
    - We achieve an overall **significant throughput improvement** over the original approach for the given benchmark, which also **includes update operations**.
5. Because the complexity of *SQL* queries for the schema can be a major hindrance, we show how to make the expressive ***Cypher* query language** (from Neo4j [Fra+18a]) **available** for the schema.

The second part of our work targets the integration of *IFC* building data as a property graph in a relational database. Therefore, we enable the future integration into the *MonArch System*. The transformation to a property graph is necessary to reduce the storage complexity: The data is represented in a very complex and not intuitive data model (containing several hundred classes and references).The interpretation as a graph reduces this complexity to the challenge of storing vertices and edges. The graph storing and querying solution proposed in this work leads to a number of improvements of the storage of *IFC* in database systems:

1. We define a **mapping** of *EXPRESS/IFC* data **into the property graph model**. Thereby, we achieve the lossless storage of *EXPRESS/IFC* data in a relational database by using the previously introduced schema.
2. The resulting system is the only known *IFC* model store that supports **arbitrary declarative queries**. Through providing the graph query language ***Cypher***, an abstract and intuitive way to construct arbitrary queries is available. No other system that stores *IFC/BIM* data provides the capability to formulate arbitrary queries on the data.
3. **Reduction of application complexity**: The data is not stored in several hundred classes and references, but as a property graph. By using the property graph model, standard graph algorithms like shortest path search can be applied. The advantages of using the property graph model have also been stated by Angles [AG08]. Therefore, very little specific knowledge about the *BIM* data format is required to start exploring and using the data, while the data is still stored lossless.
4. Storing **multiple graphs without losing efficiency** on single graphs is possible by applying horizontal fragmentation and query rewriting.
5. In contrast to other systems, like *BIMServer*, our approach supports declarative queries, including queries that **target several models** at the same time.
6. We created a **workload** to evaluate the query performance for the given IFC use case.
7. Through the use of **Geographic Information System (GIS)** capabilities offered by most DBMS, we enable **spatial queries** on the building model within the same system. The topological model is computed directly from the property graph representation of the data.

In summary, we have developed a novel approach to storing property graphs in relational database systems. We show how to make the approach usable for software development by providing algorithms for data import and enable the use of the query language *Cypher* for relational property graph stores. Afterwards, we apply the approach to a relevant specific use case. Through this application we present a novel approach to an *IFC* model server based on a relational database. This *IFC* model server offers a number of novel features compared to other stores and is highly efficient.

# Part II.

# Property Graphs in Relational Databases

# 2. Related Work

In this work we focus on the relational-based storage of property graph data. We do not consider graph processing frameworks like *Apache Giraph*[1] in our work, since we focus on the storage and retrieval of graph data and not its parallel processing. We consider solutions adequate for more general data models than RDF since, we require to assign arbitrary properties to edges.

## 2.1. Storing Graphs in Relational Database Management Systems

Since 2017, Microsoft's **SQL Server**[2] offers a feature to store graphs in their relational database system. Standard relational tables can be declared as node-tables, which implies that normal restrictions apply to those tables. Edges are also declared as tables, but standard relational restrictions regarding references do not apply to edge tables. This means source and target nodes of an edge table can be any table declared as a node table. To query the graph the *MATCH* clause can be used, but it does not support path queries of variable length, as well as queries along edges without specifying its edge type [Shk].

Another approach to store property graphs in relational databases is presented by Chen [Che13]. This approach uses arrays to represent the neighborhood of a vertex in a single table column. According to the authors this approach is well suited for sparse graphs since it avoids *NULL* values. On the other hand, Bornea et al. argue, that *NULL* values do not cause high costs in state-of-the-art RDBMS. The authors did only explore scenarios in which all vertices have a small degree. We do not consider this assumption in our work and perform an experimental evaluation with a data set that has a highly heterogeneous number of vertice degrees.

In *Edge-k* [Sca+18] Scabora et al. present a relational-based approach to store weighted graphs. They describe how to store $k$ edges of vertex in a single row. Their graph model is limited to a fixed amount of weights for an edge and can not handle arbitrary properties.

Bornea et. al first proposed the approach of shredding graph edges into adjacency tables in [Bor+13]. Sun et al. base their work in [Sun+15] on the previously mentioned work of Bornea et. al and outlined a novel schema layout for storage of property graph data in relational databases, which is generalized from the approach to store RDF data. They combine

[1] http://giraph.apache.org/

[2] https://www.microsoft.com/en-us/sql-server/sql-server-2017

the shredding of edges into adjacency tables with the use of JSON-based attribute storage to overcome the limitations of fixed columns. To make the retrieval of data more convenient, they propose a translation mechanism that converts *Gremlin* [Rod15] queries to *SQL* queries, which can be run on the proposed relational schema. The adjacency tables are the major difference between the approach by Sun et al. and the Microsoft SQL Server feature. Since **SQL Server** supports JSON data, the approach by Sun et al. could be implemented on top of the **SQL Server** graph feature. We have refined this approach for read-query heavy workloads in this thesis. A preliminary evaluation can be found in [Sch19a].

The **Apache AGE (A Graph Extension for PosgreSQL)** [Apa21] is a new extension for **PostgreSQL** for the storage of graph data. It is in the early development phase and had its first official **Apache** incubator release in early 2021. The project plans a number of features similar to our approach, like *openCypher* support, storage of multiple graphs, and queries that that target several graphs at the same time. In contrast to our approach, they currently create a table for each label type that is used. Their approach is also limited to specific versions of the database and currently only supports **PostgreSQL 11**, while the support for newer versions is currently planned. Our own approach is mostly independent of database versions, as long as the database supports JSON and array functionality. During the course of this work, we have successfully applied our approach to every **PostgreSQL** version from **PostgreSQL 10** through **PostgreSQL 13** without the need for modifications.

In particular, our approach offers a number of advantages over the storage in the native graph database **Neo4j**. First, we can store multiple graphs in the same database (or schema in our case), while in **Neo4j** the creation of several databases is required. This directly implies the second advantage: We can access several graphs with a single query. To do this in **Neo4j** we would have to create a single huge graph and distinguish between the different graphs using labels. This leads to the third advantage: we can easily and very efficiently delete single graphs.

While our general goal was to store the graph data in a RDBMS, even assuming that storing the data in e.g. **Neo4j** offers a performance advantage, our approach still comes with a number of advantages.

## 2.2. Graph Bulk Loading

The general field of graph bulk loading is largely neglected in the scientific literature, as confirmed by the few publication available (e.g. [The+16]).

In [Müh+13], Mühlbauer et al. present an approach that focuses on using all the available bandwidth while loading huge data sets over the network. Assuming that the server's main memory and hard drive speeds are not the bottleneck, they present an approach to use hardware level optimizations (e.g. CPU specific instruction sets) and highly parallelizing the import process to speed up the loading process for their relational main memory database system **HyPer**.

In their work, Then et al. [The+16] discuss that for many workloads the actual loading of the data is the dominant cost factor, yet it is not point of focus in literature. They present various loading strategies for in-memory graph representation. While we identified similar import process stages, their work focuses on in-memory representation, while our work assumes that the data set does not fit into main memory.

Durand et al. [Dur+18] examined that server-side loading is best suited for loading huge amount of data into property graph database management systems. While they mostly identified the same import phases as we did, their approach is a general concept for loading graph data. They do not consider the specific target schema, which in our case contains redundantly stored data.

## 2.3. Graph Query Languages

As stated before, we focus on storing property graph data. Therefore, we only consider graph query languages that are designed to handle queries for the property graph model or are closely related to those. Even though we store the data in a relational database, our research showed that *SQL* is not a language that is very well suited to intuitively formulate queries. The strong focus on Join operations and the table based model results in very complicated SQL queries. The last decade has brought forth a number of abstract graph query languages that have not be considered when talking about a property graph database.

In contrast to the existence of a standard query language like *SQL* [CB74] for relational databases and *SPARQL* [PAG09] for the RDF, no standardized query language for the property graph model exists [Tha+19]. Besides *SPARQL*, several other languages were proposed to query property graphs.

An early example of a graph query language is *Glide*. The authors combined features from *Xpath* and *Smart*, which resulted in a query language based on regular expressions [GS02]. In 2008, He et. al proposed *GraphQL* [HS08]. The query language is based on a graph algebra that ist extended from the relational algebra, relationally complete and contained in Datalog.

Nonetheless, none of the aforementioned query languages has managed to assert itself as a standard query language, yet. Currently, three languages have come closest to becoming the standard query language for property graphs: *Gremlin* is a low level graph traversal language developed by *Tinkerpop* [Tha+19]. Several graph databases like **Neo4j** and **OrientDB** offer *Gremlin* interfaces. Even though many systems offer an interface for the use with the *Gremlin* query language, many application developers prefer other languages like *Cypher* [HP13] due to its non-declarative nature.

Created by *Facebook*, *GraphQL* [Fac16] (not to be confused with the aforementioned GraphQL [HS08]) is a conceptual framework for providing a graph data access interface on the Web. This framework includes a graph query language that was first formally specified by Hartig et al. [HP18]. To write a query, the user defines a JSON-like skeleton including

so-called fields that can contain values. The fields that are defined but not bound with values are then filled by the query engine with corresponding result values [HP17].

The *Cypher* graph query language was originally developed by the **Neo4j** team and is a declarative graph query language [Fra+18a]. The main operator of the query language is the **MATCH** which is used to describe the graph pattern that we are searching for, much like the way we describe patterns in *SPARQL*. The *Cypher* query language has moved on to the openCypher project [Gre+18].

## 2.4. Graph Benchmarks

In the field of *RDF* there exist numerous benchmarking efforts. But to the best of our knowledge the benchmarks provided by the *Linked Data Benchmark Council (LDBC)* are the only benchmarks that directly address the problem of benchmarking property graph stores.

The *LDBC* is an EU project with the goal to develop benchmarks for graph structured data. They strife to find the acceptance benchmarks like the TPC [PF00] have achieved. The *LDBC-SNB* is a benchmark framework being developed by the *LDBC* that uses a generated social network graph as its data set and represents the data as a property graph. The benchmark suite provides a data set generator that can create test data with different scale factors.

In contrast to most generated graphs, the generator tries to generate data as realistically as possible. To this end different strategies to correlate attribute values and the graph structure are applied. For example two people studying at the same university are more likely to know each other, than people who have no common place. An overview of different scale factors and the corresponding number of vertices and edges is depicted in Table 2.1.

| SF | Approx. #Vertices | Approx. #Edges |
|---|---|---|
| **1** | 3,200,000 | 17,300,000 |
| **3** | 9,300,000 | 52,700,000 |
| **10** | 30,000,000 | 176,600,000 |
| **30** | 99,400,000 | 655,400,000 |
| **100** | 317,700,000 | 2,154,900,000 |
| **300** | 907,600,000 | 6,292,500,000 |

Table 2.1.: Approximate number of vertices and edges generated for different LDBC-SNB scale factor (SF).

Works on the *Social Network Benchmark* are not finished yet, but the *Interactive Workload* has been released in draft stage [Erl+15]. This workload can be partitioned into three types of queries: First, *simple queries* request a small part of the graph, while most of the time starting at a single vertex and only considering a small neighbourhood around this vertex. Second, *complex queries* take a big part of the graph into account and often include complex

aggregations. Last, *update queries* create new vertice(s) and insert edges connecting those new vertices.

In contrast to relational databases and RDF triple stores, no de facto standard query language for property graph stores exists. Therefore, all queries are defined in natural language in a specification document [Ang+20]. Naturally this introduces a source for variance, depending on the quality of the implementation of the queries for the system being benchmarked. Other workloads, like the business intelligence workload, are currently under development [Szá+18].

# 3. The Relational Property Graph Model

In this chapter we describe our storage approach for huge property graphs in relational database managements systems.

As we described in Section 1.1, we were motivated to store property graph data in a RDBMS due to our use case to integrate BIM data into the *MonArch* system [FS17]. Based on a preliminary feasibility study, we had shown the advantages of storing IFC data as a property graph. We validated this by implementing a proof of concept using **Neo4j**. Nevertheless, the requirements of the BIM use case required integrating the data into a relational database (in this specific case we used **PostgreSQL**).

We first present the definition of a property graph we used for our work. Then we introduce our database schema. Afterwards we propose a set of indices for use with **RATG**. We finish the chapter with a section describing how the database schema can be used for data querying, data insertion and deleting data.

## 3.1. Property Graphs

Many definitions of a property graph can be found in literature, but to the best of our knowledge there is no generally accepted definition, though most of the definitions only differ in minor details. For this thesis we use the definition given by Angles [Ang18]:

We assume that $L$ is an infinite set of labels, $P$ is an infinite set of property names (which are also called keys), and $V$ is an infinite set of atomic values.

**Definition 3.1** [Property Graph]
A **property graph** consists of a **set of vertices** and **directed edges**. Every edge has a **label** assigned to it. Each vertex or edge can additionally have **multiple key-value pairs** assigned that serve as *attributes*. More formally, a **property graph** is a tuple $G = (N, E, \rho, \lambda, \sigma)$ where:

1. $N$ is the finite set of nodes (also called vertices);

2. $E$ is the finite set of directed edges such that $N \cap E = \emptyset$;

3. $\rho : E \to (N \times N)$ is a total function that associates each edge $r \in E$ with a pair of nodes $n_1, n_2 \in N$ (therefore $\rho$ is the well-known incidence function from graph theory). We also call $n_1$ the source node and $n_2$ the target node of the edge $r$.

   Additionally, for the sake of readability we define $src : E \to N$ as the function that associates the source vertex $n_1$ with the edge $r$ and $target : E \to N$ as the function that associates the target vertex $n_2$ with the edge $r$;

4. $\lambda : E \to L$ is a total function that associates an edge with a label $l \in L$. We also call this label the type of the edge;

5. $\sigma : (N \cup E) \times P \to \mathscr{P}(V)$ is a partial function that associates nodes and edges with properties. For each property it assigns a set of values from $V$. As in [Fra+18b], we assume that $\sigma$ is a total function, but there are only finitely many $x \in (N \cup E)$ and $k \in P$ such that $\sigma(x,k) \neq null$.

□

**Remark 3.1**
We use a reserved property key "type" to assign types to vertices. For the sake of a clear depiction, we will color the text representation of vertex types blue (e.g. person), of relationship types green (e.g. knows), of vertex properties orange (e.g. *name = "Yamamoto"*), and of edge properties red (e.g. *since = "14.06.2018"*). We will use this highlighting for the remainder of this thesis. □

We will base most examples given in this thesis on the same small example graph. Therefore, we now introduce this graph.

**Example 3.2** [Example Graph for this Thesis]
Figure 3.3 depicts an example of a simple property graph that describes a social network graph. The graph describes three people (nodes identified by **1**, **2** and **3**). The *person* with the *name = "Yamamoto"* *knows* the *person* with the *name = "Silva"* *since = "14.06.2018"*, which is described by the directed edge between the two vertices. The nodes identified by **1** and **2** like the node identified by **7** that describes a *post*, but no attributes are assigned to those edges. The rest of the example can be interpreted analogously.

This graph is formally represented in the previously presented model as $G = (N, E, \rho, \lambda, \sigma)$ with

- $N = \{n_1, n_2, n_3, n_7, n_{13}\}$ as the set of nodes;
- $E = \{r_4, r_5, r_6, r_{11}, r_{12}, r_{14}\}$ as the set of edges;
- $\rho = \{r_4 \to (n_1, n_2), r_5 \to (n_1, n_3), r_6 \to (n_2, n_7), r_{11} \to (n_{13}, n_1), r_{12} \to (n_1, n_7),$ $r_{14} \to (n_{13}, n_7)\}$ as the total function that associates each edge with its corresponding source and target vertex;

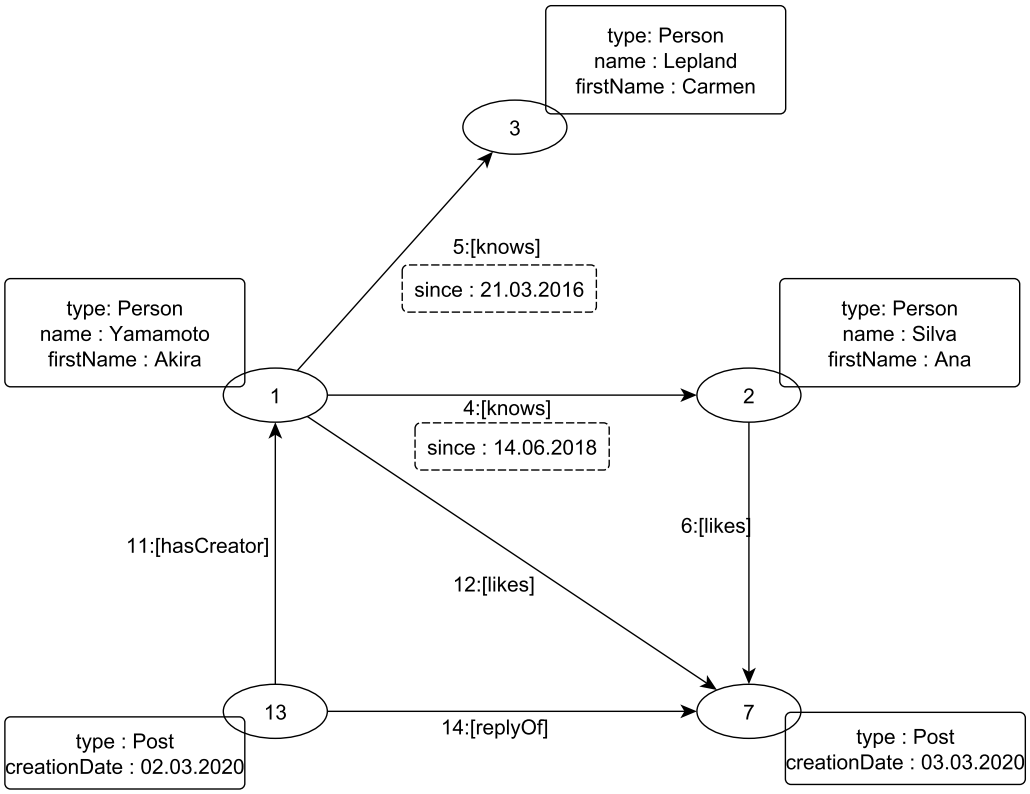



Figure 3.3.: A property graph example modeling a social network.

- $src = \{r_4 \to n_1, r_5 \to n_1, r_6 \to n_2, r_{11} \to n_{13}, r_{12} \to n_1\}$ as the total function that assigns each edge its source node;
- $target = \{r_4 \to n_2, r_5 \to n_3, r_6 \to n_7, r_{11} \to n_1, r_{12} \to n_7\}$ as the total function that assigns each edge its target node;
- $\lambda = \{r_4 \to knows, r_5 \to knows, r_6 \to likes, r_{11} \to hasCreator, r_{12} \to likes, r_{14} \to replyOf\}$ as the total function that assigns a label to each edge;
- and
  $\sigma(n_1, type) = Person$,
  $\sigma(n_1, name) = Yamamoto$,
  $\sigma(n_1, firstName) = Akira$,
  $\sigma(n_2, type) = Person$,
  $\sigma(n_2, name) = Silva$,
  $\sigma(n_2, firstName) = Ana$,
  ...
  $\sigma(r_4, since) = 14.06.2018$,

$\sigma(r_5, since) = 21.03.2016$
as the partial function that assigns properties to vertices and edges.

□

Note that we have to represent undirected edges by using exactly two directed reciprocal edges. To avoid the definition of an additional function, we assume that comparison operators like $<$ can be applied to two vertices (or edges respectively) to compare the IDs of vertices (and edges respectively), for example $n_1 < n_2$ and $n_2 \geq n_1$. We will use this example graph as a running example for the rest of this part of the thesis.

## 3.2. The Database Schema

Our goal was to find a storage schema for property graph data in a relational database with an emphasis on supporting efficient read-only queries. We apply relational data storage and combine it with the JSON functionality of relational DBMS, which is by now available in almost every relational DBMS.

We store edges redundantly in an edge list table and two tables that represent incoming and outgoing adjacency lists of the vertices. We achieve high read-query performance, because the neighborhood of a vertex can usually be found by loading a single tuple of the adjacency table. Our work significantly improves the efficiency of the work described in [Sun+15]. An overview of the tables included in our schema is depicted in Table 3.4.

Vertices : {[
    VID : LONG,
    Attributes : JSON
]}

Edges : {[
    EID : LONG,
    SID : LONG *(→ Vertices.VID)*,
    TID : LONG *(→ Vertices.VID)*,
    Label : VARCHAR,
    Attributes : JSON
]}

OutgoingAdjacency : {[
    VID : LONG *(→ Vertices.VID)*,
    $EID_1$ : LONG[],
    $Label_1$ : VARCHAR,
    $TID_1$ : LONG[],
    ...
    $EID_k$ : LONG[],
    $Label_k$ : VARCHAR,
    $TID_k$ : LONG[]
]}

IncomingAdjacency : {[
    VID : LONG *(→ Vertices.VID)*,
    $EID_1$ : LONG[],
    $Label_1$ : VARCHAR,
    $SID_1$ : LONG[],
    ...
    $EID_p$ : LONG[],
    $Label_p$ : VARCHAR,
    $SID_p$ : LONG[]
]}

Table 3.4.: The relational property graph database schema.

We will now describe the different tables including our optimization for read-only focused workloads. For the storage of vertex data and the storage of edge attributes we apply the concept presented in [Sun+15].

**Storing Vertices** The **vertex table** stores the internal vertex identifier and the attributes for each vertex. The internal vertex identifier VID is automatically generated at load time and solely used for internal referencing purposes. It should not be used to represent actual data. In our prototypical implementation the attributes are implemented as the *jsonb* data

type provided by **PostgreSQL**. To describe vertex types we can *reserve* a key for the *jsonb* document.

An example for the *vertex table* is depicted in Table 3.5: For our evaluation data set (see Section 6.1) we reserved the property key *type* for the label of a vertex, in order to model vertex types like Person, Post, and Message. The table also contains other attributes like the *name* of a Person and the *creationDate* of a Post. For example, the vertex $VID = 3$ is of type Person and has the *name = "Lepland"*.

| VID | Attributes |
|---|---|
| 1 | {"type": "Person", "name": "Yamamoto", "firstName": "Akira"} |
| 2 | {"type": "Person", "name": "Silva", "firstName": "Ana"} |
| 3 | {"type": "Person", "name": "Lepland", "firstName": "Carmen"} |
| 7 | {"type": "Post", "creationDate": "03.03.2020"} |
| 13 | {"type": "Post", "creationDate": "02.03.2020"} |

Table 3.5.: The vertex attributes table of the graph in Figure 3.3.

**Storing Edges as an Edge List** We store **edge attributes** analogously. In contrast to the vertex table, we do not only store attributes in this table, but also additional information about the edges like the source vertex SID and the target vertex TID. Since in our definition of a property graph (see Definition 3.1) every edge has exactly one label, we also store these in a designated column. In [Sun+15] it is claimed that queries which require tracing along a variable number of edges are significantly more efficiently computed when the edge table is used instead of the adjacency table. We performed a preliminary evaluation supporting this claim [Kor17].

| EID | SID | TID | Label | Attributes |
|---|---|---|---|---|
| 4 | 1 | 2 | knows | {"since" : "14.06.2018"} |
| 5 | 1 | 3 | knows | {"since" : "21.03.2016"} |
| 6 | 2 | 7 | likes | null |
| 11 | 13 | 1 | hasCreator | null |
| 12 | 1 | 7 | likes | null |
| 14 | 13 | 7 | replyOf | null |

Table 3.6.: The edge attributes table of the graph shown in Figure 3.3.

An example for the *edge list* is depicted in Table 3.6: The edge with $EID = 5$ has the source vertex with $VID = 1$, target vertex with $VID = 3$, the relation type knows and the attribute value ”21.03.2016” for the key *since*.

We assumed that in most graph applications queries will most commonly access a huge part of the neighborhood of a vertex and not just single edges. Quite often these queries are restricted to a set of edge labels, but rarely access single edges. We want to leverage this property by

storing edges in such a way that they can be quickly accessed. To this end we apply the concept of adjacency lists to our database schema.

**Storing Edges Using Adjacency Tables** Our solution **differs** from **SQLGraph** in respect to how **adjacency tables** are implemented. In their work, Sun et al. [Sun+15] claim that shredding vertex adjacency lists into a relational schema provides a significant advantage over other mechanisms, for example mechanisms that store all edge information in a single table. To this end a hash function has to be defined that matches edge labels to a corresponding column triple of the adjacency table.

In addition, we **generalize** the approach to compute a hash function through the use of coloring heuristics as described in [Bor+13] for data models. Note that, since the LDBC-SNB data set includes a data model, it is possible to compute a conflict-free hash function for this specific data set. We will present our approach to construct this hash function in Section 4.1.

Sun et al. [Sun+15] store adjacency data in two types of tables, namely the primary adjacency tables and secondary adjacency tables. If only a *single* edge of a label exists for the vertex, its edge-id EID, label and target vertex-id TID are stored in the *outgoing primary table (OPA)* and *incoming primary table (IPA)*, respectively. If a vertex is only the source of each **edge type** a **single time**, then **all** outgoing **neighbors** of the vertex can be found by simply retrieving the **single tuple** that stores all of the neighbors. According to Sun et al. [Sun+15] this leads to a significant performance improvement, even in comparison to native graph databases (like for example **Neo4j**) for these kinds of data sets.

However, by studying available data sets and use cases one can easily see that single occurrences of edges are usually not the case. For example, consider the LDBC-SNB data set that we used for our performance evaluation (see Chapter 6): The data set mainly contains edges like knows, isLocatedIn, worksAt, hasCreator, etc. all of which are relationships that can target or originate from several vertices.

**OPA**

| VID | $EID_1$ | $Label_1$ | $TID_1$ | ... | $EID_k$ | $Label_k$ | $TID_k$ |
|---|---|---|---|---|---|---|---|
| 1 | 12 | likes | 7 | | null | knows | -1 |
| 2 | 6 | likes | 7 | | null | null | null |
| 13 | 11 | hasCreator | 1 | | 14 | replyOf | 7 |

**OSA**

| TMP | EID | TID |
|---|---|---|
| -1 | 4 | 2 |
| -1 | 5 | 3 |

Table 3.7.: The outgoing adjacency tables of the original **SQLGraph** for the graph in Figure 3.3.

Therefore, if the vertex has multiple outgoing edges of the same label, the edge-ids and target vertices must be stored in the *outgoing secondary adjacency table (OSA)* and *incoming secondary adjacency table (ISA)* respectively, while the target vertex id of the *outgoing primary table* or *incoming primary table* is set to a generated abstract value that serves as the JOIN partner for the *outgoing secondary adjacency table* and *incoming secondary adjacency table* [Sun+15].

An example for multiple outgoing edges with the same label is depicted in Table 3.7: In our example graph the vertex with $VID = 1$ knows the two vertices with $VID = 2$ and $VID = 3$. This leads to both edges being outsourced to the *secondary adjacency* table. In turn, this means that in order to retrieve the neighborhood of a vertex using the *adjacency tables*, the user needs to use a potentially expensive JOIN operation.

The example also demonstrates the **two major drawbacks** of **SQLGraph**:

1. A query to receive all outgoing or incoming neighbors requires the use of an OUTER JOIN operation. This is necessary, since the user cannot be sure if the data is outsourced to the *secondary adjacency table*. In order to provide a query that works in any case, this JOIN is **mandatory**. If there exists only a single edge, the OUTER JOIN will not find a corresponding partner in the *secondary adjacency* table and therefore use the data in the *primary adjacency* table. If more edges do exist, the JOIN results will represent the resulting edges. This query works independently of the number of edges per label and the hash function. Therefore, for every edge-hop a **potentially expensive** JOIN operation is required. Because retrieving the neighborhood of a vertex is one of the most essential operations for graph pattern queries, this will occur in nearly every query. Our evaluation (see Section 6.3) confirms the statement from [Sun+15] that big intermediate result sets are a bottleneck for their approach.

2. The unoptimized approach assumes that many of the edges **only occur once** for a vertex. But for every type of edge that occurs $n > 1$ times, the respective *secondary adjacency* table contains $n$ tuples. Therefore, in a data set that mainly contains edges that **primarily occur several times**, the corresponding *secondary adjacency* table is nearly the **same size** as the **edge list** table. For the worst case, the *secondary adjacency* table contains exactly the same amount of tuples as the *edge list* table, which in turn **defeats** the original **purpose** of shredding the edges in the *primary adjacency* table, thereby reducing the search space to retrieve the neighborhood of a query.

We **eliminate** the need for this OUTER JOIN-operation, and thus **optimize** query evaluation by storing all edges of one type in the corresponding columns using arrays (e.g. **PostgreSQL** offers a JSON-array data type which can be used to implement the proposed schema). We trade the overhead of always performing an OUTER JOIN-operation for the cost of unwrapping the values of arrays. Doing this, we **overcome** the **bottleneck** of always having to perform the aforementioned OUTER JOIN operations. We can confirm this using our evaluation results (see Section 6.3)

Assuming we have found a hash function with a very low amount of conflicts, the number of entries in the adjacency tables is very close to the number of vertices of the graph. This is due to the fact that without hash conflicts we can store the complete neighborhood of a vertex in a single tuple. The number of vertices in a graph is usually much lower than the number of edges in the graph and therefore our adjacency tables have a lot less entries than the secondary adjacency entries. Therefore, we **greatly reduce** the search space required to find the neighborhood of a vertex for most data sets.

| VID | $EID_1$ | $Label_1$ | $TID_1$ | ... | $EID_k$ | $Label_k$ | $TID_k$ |
|---|---|---|---|---|---|---|---|
| 1 | [12] | likes | [7] | | [4, 5] | knows | [2, 3] |
| 2 | [6] | likes | [7] | | null | null | null |
| 13 | [11] | hasCreator | [1] | | [14] | replyOf | [7] |

Table 3.8.: The outgoing adjacency table for the graph in Figure 3.3.

Table 3.8 shows an example of our adjacency tables: The graph contains two knows edges with edge identifiers $EID = 4$ and $EID = 5$ that point from the vertex with $VID = 1$ to the vertices $VID = 2$ and $VID = 3$. The complete outgoing neighborhood of the vertex can be stored in a single tuple.

| VID | $EID_1$ | $Label_1$ | $SID_1$ | ... | $EID_p$ | $Label_p$ | $SID_p$ |
|---|---|---|---|---|---|---|---|
| 1 | null | null | null | | [11] | hasCreator | 13 |
| 2 | [4] | knows | [1] | | null | null | null |
| 3 | [5] | knows | [1] | | null | null | null |
| 7 | [12, 6] | likes | [1,2] | | [14] | replyOf | [13] |

Table 3.9.: The incoming adjacency table belonging to the graph depicted in Figure 3.3.

Table 3.9 depicts an example for an incoming adjacency table for the graph in Figure 3.3. The two knows edges with $EID = 4$ and $EID = 5$ can be found in the respective tuples for the vertices with $VID = 2$ and $VID = 3$, because these are the target vertices of the edges. In particular, it should be noted that the hash function for the incoming adjacency table is not the same hash function as is used for the outgoing table. For example, in this case it is possible to map both the knows and the likes edges to the first column triple, since these types of edges always target different types of vertices (Person and Post, respectively). Not only can the assignment of labels to column triples be different from the outgoing hash function, but the number of required columns can divert from the amount of the outgoing table, too.

However, if two labels are hashed to the same column triple, we need to insert a second row for this vertex. For examples sake, let us assume that the labels knows and likes of our example graph (see Figure 3.3) are both mapped to the first column triple of the outgoing adjacency table. The result of inserting both types of edges into the table can be seen in Table 3.10: Since the first column triple is already filled with likes edges and the given hash function also maps the knows edges to the first column triple, we have to insert a second tuple for the vertex with $VID = 1$ to be able to also store edges of the type knows for this vertex.

| VID | $EID_1$ | $Label_1$ | $TID_1$ | ... | $EID_k$ | $Label_k$ | $TID_k$ |
|---|---|---|---|---|---|---|---|
| 1 | [12] | likes | [7] | | ... | ... | ... |
| 1 | [4, 5] | knows | [2, 3] | | null | null | null |
| 2 | [6] | likes | [7] | | null | null | null |
| 13 | [11] | hasCreator | [1] | | [14] | replyOf | [7] |

Table 3.10.: The outgoing adjacency table depicted in Table 3.8 with a hash conflict highlighted in red for the labels knows and likes.

We could omit this by providing a conflict-free hash function or possibly by providing more columns.

An evaluation of optimal numbers of columns has not been part of our research yet and will have to be addressed in the future. If no hash function with a low number of resulting conflicts for a given amount of columns can be found, additional columns could be required. Nevertheless, we were able to find such a hash function for our IFC use case (see Chapter 7), as well as for the LDBC-SNB data set [Bon13], which we used for our evaluation. We will present our approach to compute such a conflict-free hash function in Section 4.1 and present our evaluation results in Chapter 6.

In conclusion, we **eliminate** the need for the OUTER JOIN-operation by **storing** all **neighbors in arrays** instead of a reference to a *secondary adjacency table*. We expected the overhead we create by using arrays to be much lower than the overhead created by performing the JOIN-operations necessary to use **SQLGraph**. This even holds true for the existence of single edges, which is the strong point of the approach. By eliminating the need of an OUTER JOIN for any edge hop, a **major drawback** of the original schema is **overcome**. We present our findings regarding the performance of our optimization in Chapter 6.

### 3.2.1. Indexing the Graph

As in any application of relational databases, the use of indexes to access the data is vital to achieve good performance. Since the stored graphs are huge, and for example searching for specific nodes needs to be done in the shortest amount of time possible, indexes relieve the database system of loading all data from the hard drive and therefore help to narrow down the search. The utilized indexes in our proposed approach **RATG** can be split into two types:

1. Indexes that will be useful in any application regarding the property graph data. Those indexes offer a general improvement of access time for the graph data that is independent of the use case at hand. They enable the fast search for vertices, access to the vertices' attributes and associated edges. In this section we describe the indexes we propose to use for any application.
2. Indexes that are specific to a certain use case. For example, in the LDBC-SNB a very frequent query searches for a node with a specific *id* that is only unique in combination with the *type* of the node. A specialized index that contains the *id and type* of vertices can speed up query processing significantly. Examples for use case specific indexes for the LDBC-SNB can be found in Section 6.2.2 and for the IFC use case in Section 10.2.3.

We now describe the set of indexes we propose to always use in combination with our approach.

The **vertex table** is indexed as follows:

***Index* on VID** Any time attribute values regarding a vertex that is source or target of an edge are requested, an Equi-Join between the vertex table and source/target id of the edge based on the id must be performed. Since accessing the neighborhood of a vertex is an essential operation in a graph, we index the global graph vertex id using an index. We do not expect range queries (which would make a B-Tree preferable) on the vertex id, since this id is strictly used for internal referencing mechanisms and should not carry application information.

***Inverted Index* on attributes** We index the attributes of a node with an inverted index. This enables us to search for vertices using attribute values efficiently. If the expected workload targets specific attributes, we suggest creating additional indexes.

Then we index the **edge list** as follows:

***Hash Index* on EID** We need to use an Equi-Join of edge list table with intermediate query results every time we need to access edge attributes. Because no range queries should occur on this attribute, we propose to use a hash index.

***Hash Index* on SID** As mentioned before, outside of accessing attributes of edges the edge list is mainly used for recursive SQL queries to traverse paths of variable length. These recursive queries once again heavily use Equi-Joins. To speed up this process we propose a hash index on the source vertex id of the edges.

***Hash Index* on TID** Due to the same reason, we propose a hash index on the target id of an edge. This speeds up recursive queries in the opposite direction.

Lastly we index the **outgoing adjacency** table and **incoming adjacency** table the same way:

***Hash Index* on VID** For most edge hops in the graph we will join the outgoing adjacency table or incoming adjacency table to the current intermediate result by use of an Equi-Join. Therefore, we propose to use a hash index on the vertex id of both tables.

These indexes are the base for the graph schema. Additional workload-specific indexes can speed up query processing further by, for example, enabling the DBMS to answer (sub-)queries using index-only-lookups. Examples for these application-specific indexes can be found in Section 6.2.2.

### 3.2.2. Using the Database Schema for Graph Queries

Now that we have introduced our database schema, we will take a brief look at how to retrieve and manipulate graph data stored in our schema using *SQL* for data retrieval and *Procedural Language/PostgreSQL Structured Query Language (PL/pgSQL)* for data modification. For this purpose, we will use examples written in the **PostgreSQL** dialect. While the implementation for this work mainly focused on **PostgreSQL**, Shanmugam has also created a proof of concept for **OracleDB** [Sha21].

**Retrieving Data** In order to construct queries that represent graph pattern queries, we have to be able to perform three basic graph queries: Retrieving a vertex and its attributes (e.g. using the vertex id or attribute values), retrieving an edge and its attributes, and the neighborhood of a vertex (for either outgoing or incoming edges). All other types of queries can then be constructed from these basic operations using Common Table Expression (CTE)s and (OUTER) JOIN operations.

Listing 3.11 depicts a query that searches for vertices using the type person and that has the *name* 'Yamamoto'.

```
SELECT a.vid, a.attributes
FROM vertices a
WHERE a.attributes->>'type' = 'person'
  AND a.attributes->>'name' = 'Yamamoto'
```

Listing 3.11: Example query that searches for a Vertex.

Querying for vertices and their attributes can be achieved using standard *SQL* and JSON functions provided by the DBMS. Analogously, the same can be achieved for edges and their attributes.

To be able to apply the concept of adjacency lists, we now consider the query depicted in Listing 3.12: This query searches for the outgoing neighborhood of the vertex with $VID = 3$, but only edges of type likes.

The first sub-query *unshred_edges* retrieves the relevant adjacency lists for the vertex with $VID = 3$. If we have a hashing function available that produces a low amount of conflicts, the number of rows that have to be loaded is very low. By applying the combination of UNNEST and array functions, we split the single row that contains $k$ column-triples into $k$ rows that each contain a single column-triple.

The next sub-query *gather_edges* uses the DBMS specific function *array_elements*() to split each of the rows, which contains a single column-triple, which in turn contains arrays of length $n$ into $n$ rows of which each contains a single edge. The results are filtered to remove any edges that do not have the label likes.

```
WITH unshred_edges AS (
  SELECT vid,
    UNNEST(array[EID_1,...,EID_k]) AS eidTmp,
    UNNEST(array[Label_1,...,Label_k]) AS label,
    UNNEST(array[TID_1,...,TID_k]) AS tidTmp
  FROM OutgoingAdjacency
  WHERE vid = 3
),
gather_edges AS (
  SELECT unshred_edges.vid,
    array_elements(eidTmp) AS eid,
    label,
    array_elements(tidTmp) AS target
  FROM unshred_edges
  WHERE label = 'likes'
),
SELECT vid, eid, label, target
FROM gather_edges
```

Listing 3.12: Example query to search for the outgoing neighborhood of a vertex.

**Remark 3.2**
In the course of this work we will use abbreviated versions of **PostgreSQL** functions to make the listings and examples more readable. The queries can not be used in this exact form. We also use the *jsonb* array instead of the original **PostgreSQL** array data type, since more predefined functions are provided for *jsonb* arrays. □

**Remark 3.3**
For a detailed description of how to construct complex *SQL* queries that are equivalent to complex *Cypher* queries, please consider Section 5.2.1. □

**Inserting Data** In order to insert new vertices into the graph, we can use standard *SQL* update queries. Therefore, we will only take a closer look at the creation of new edges.

Since **RATG** leverages redundant storage of edges to improve query performance, we need to make sure that each new edge is inserted three times. To this end we assume that we already know the *vid*s of the source and target vertex, the label and the corresponding column-triple numbers for the incoming and outgoing adjacency tables.

1. The edge has to be inserted into the edge list table. We can do this using standard *SQL*. Doing this we can also let the DBMS handle the generation of an edge id;

2. Using the edge id, source vertex id and target vertex id, we need to insert the edge into the outgoing adjacency list of the source vertex;

3. We need to analogously insert the edge into the incoming adjacency list of the target vertex.

We assume that we have a hash function that maps each edge to a column triple. We describe how to obtain such a mapping in Section 4.1.

To do this we implemented prepared functions in **PostgreSQL** using **PL/pgSQL**. An abbreviated example of the function that inserts an edge into the graph is depicted in Listing 3.13.

**Deleting Data** Since deleting data was not part of our IFC use case, we did not focus on the evaluation of vertex and edge deletion.

In order to delete an edge, we have to make sure that the following three actions are performed:

1. The edge is removed from the edge list table. We can achieve this using standard *SQL*;

2. The edge is removed from the outgoing adjacency table of the source vertex. We can do this by performing a similar query as depicted in Listing 3.13, but removing the value from the array instead of appending it;

3. And the edge is removed from the incoming adjacency table of the target vertex. This can be done analogously to deleting data from the outgoing adjacency table.

When we want to delete a vertex we have to perform the following steps:

1. The vertex is removed from the vertex table. We can easily achieve this by performing a standard *SQL* query;

2. All edges with the removed vertex are deleted:

    a) All edges that contain the vertex are removed from the edge list using standard *SQL*;

b) We change the id $x$ of the outgoing adjacency list entry for the given vertex to $-1 * x$; this means we do not have to remove the edge entries from the edge target's incoming adjacency list right now. These edges will from now on not be included in query results, since they will not successfully join on existing vertices. A maintenance job can clean up those remaining entries later on when the system is not under stress.

c) And we remove the incoming adjacency list entries for the given vertex using the same approach as we use for the outgoing adjacency list.

Up to this point we have assumed that we already know the hash functions for the incoming and outgoing adjacency tables. We will now present how to compute suitable hash functions.

```
CREATE OR REPLACE FUNCTION insertEdge(source bigint, target bigint,
  label text, attributes text, incolumn integer, outcolumn integer)
  RETURNS bigint
  LANGUAGE plpgsql
AS
$function$
  DECLARE
    generatededgeid bigint;
    insert_opa_command text := 'UPDATE␣outgoingAdjacency␣SET␣eid'
      || outcolumn ||'␣=␣COALESCE(eid' || outcolumn ||
      ',␣jsonb_build_array())␣||␣jsonb_build_array($4),'
      ...
      'WHERE␣vid␣=␣$1␣AND␣InsertTuple␣=␣true';
    insert_ipa_command text := ...
  BEGIN
    INSERT INTO edges (incomingedge, outgoingedge, label,
      edgeattributes) VALUES ($1, $2, $3, $4::jsonb)
      RETURNING edgeid INTO generatededgeid;
    -- ensure that adjacency tuples for the source vertex exist
    INSERT INTO outgoingAdjacency (vertexid, InsertTuple)
      SELECT SOURCE, true
      WHERE NOT EXISTS (
        SELECT 1
        FROM outgoingAdjacency
        WHERE vid = SOURCE
      );
    INSERT INTO incomingAdjacency (vertexid, InsertTuple)
      SELECT SOURCE, true
      WHERE NOT EXISTS (
        SELECT 1
        FROM incomingprimaryadjacency_1
        WHERE vid = SOURCE
      );
    -- repeat same for target vertex
    ...
    -- append edge triple to the corresponding arrays
    EXECUTE insert_opa_command USING SOURCE, target, label,
              generatededgeid;
    EXECUTE insert_ipa_command USING SOURCE, target, label,
              generatededgeid;
    RETURN generatededgeid;
  END;
$function$;
```

Listing 3.13: Example implementation of a prepared function to insert a new edge.

# 4. Data Import

The time-efficient import of huge amounts of data is imperative and a challenge for every database system. In the case of huge data graphs additional challenges arise: Most use cases for graph data consider particularly huge amounts of data, while at the same time data locality in graphs is a particularly difficult topic. Our approach stores the adjacency of a vertex in a single row. Since we want to minimize random accesses to the permanent storage, as well as round trip times to the database, we need to gather all edges of a vertex before we can create its adjacency entry. This also limits streaming capabilities for the data import and calls for specialized import strategies.

In this chapter we present an approach that describes how to compute suitable hash functions that are based on a given data model. At the same time the number of required columns to store the data is computed, such that we can avoid any conflicting hash values, and therefore the need to overload columns. Afterwards, we show our approach of how to import huge amounts of data efficiently into **RATG** using the previously described hash functions.

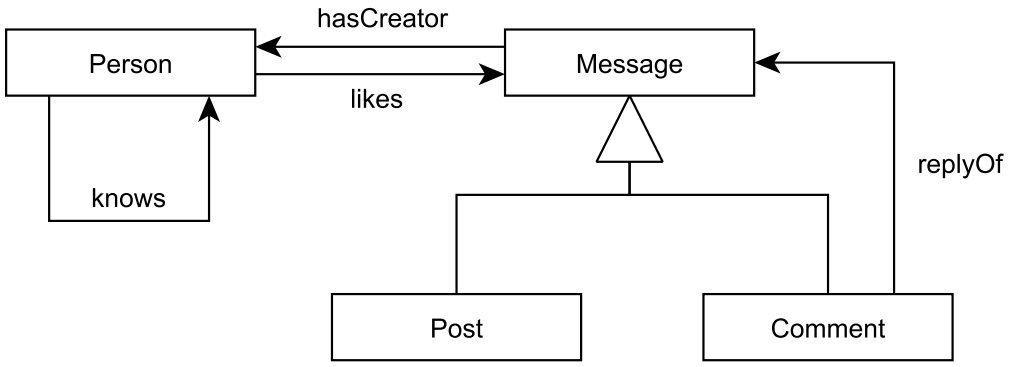



Figure 4.1.: LDBC-SNB data model excerpt.

Since our approach relies on the presence of a data model, we first need to clarify our understanding of a data model for this work. With the term **data model** we refer to the entity types that are present in the structure of the data, as well as the relationship types that exist between those entities and the direction of the relationships. We use a **UML** class diagram style notation, but omit the specification of attributes and methods, since those are not necessary, nor relevant to our approach. An example for this illustration is depicted in Figure 4.1. We decided to use **UML** diagrams over **Entity-relationship** models, because the

direction of the relationships is relevant for our approach and most readers should be familiar with this type of diagram.

In contrast to the data model, a data instance in accordance to a given data model is the actual data that follows the structure defined by the model. An example for a data instance that follows the structure defined by the data model in Figure 4.1 is depicted as a property graph in Figure 4.3.

Our **approach** is **model-based**. This means our approach relies solely on the use of a given data model. Therefore, we can initialize the schema and everything necessary (e.g. hash functions) to start processing data, before we have any data instances. In contrast to this an instance-based approach (like [Sun+15]) uses data instances as the basis for any decision.

We extend the work of Sun et al., who did not present an approach that shows how data can efficiently be imported into their database schema. In their work they use data instances to derive a mapping of edges to corresponding columns. Nevertheless, they do not present how to determine a suitable number of columns, which is necessary to use the approach.

Note that we present our approach based on **RATG**, but it can also easily be applied to the approach of Sun et al [Sun+15].

## 4.1. Hash Functions Based on Data Models

To use the previously presented approach (see Section 3.2), we require hash functions for the outgoing and incoming adjacency tables respectively.

**Definition 4.2** [Hash Function from Label to Column]
Let $L$ be the set of labels as defined in Definition 3.1. Then

- $\delta_{in} : L \to \mathbb{N}_0$ is the function that associates each edge type with a natural number for the incoming adjacency table, and analogously
- $\delta_{out} : L \to \mathbb{N}_0$ is the function that associates each edge type with a natural number for the outgoing adjacency table.

The associated numbers represent the column triple. We will store this type of edge for the incoming adjacency table and outgoing adjacency table, respectively. □

**Example 4.4**
Recalling the example graph from Section 3.1 here depicted in Figure 4.3, we can see that edges of the two types likes and knows, as well as hasCreator and replyOf, never have the same vertex as a target.

If we find a way to confirm this fact, we could store those two types of edges in the same column, since no vertices are the target of both. Therefore, the hash function would never create any conflict for the incoming adjacency table regarding these edge types. Assuming

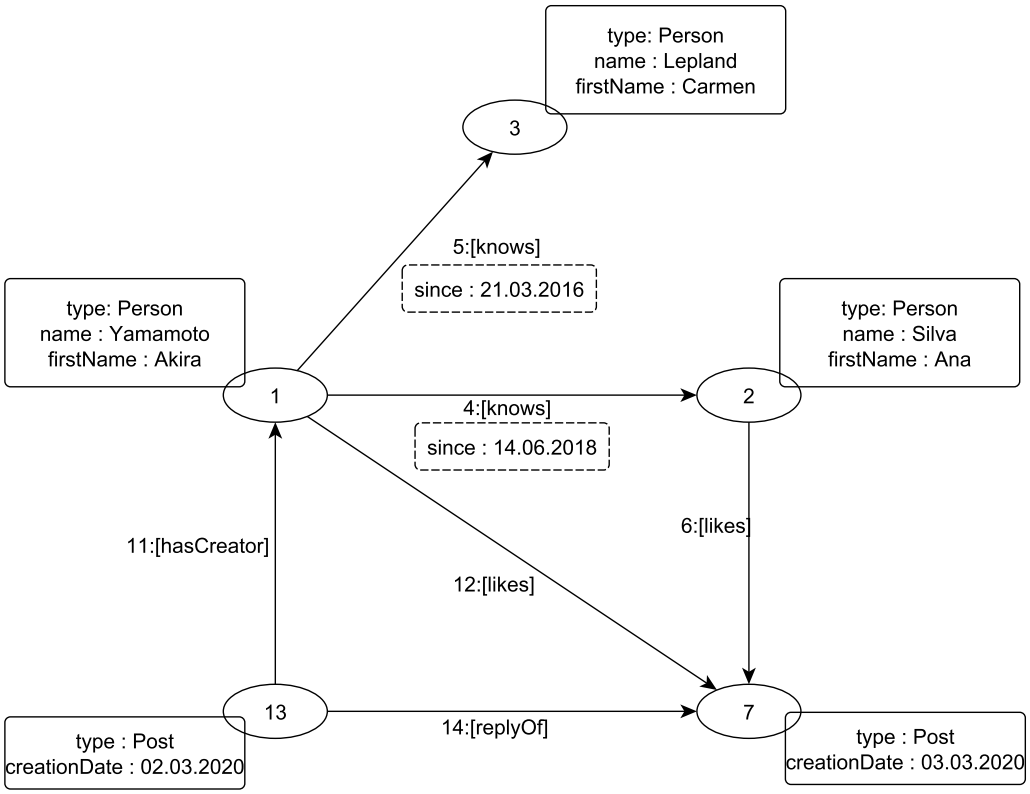



Figure 4.3.: A property graph example modeling a social network.

this fact holds, a suitable hash function could be
$\delta_{in} = \{knows \to 0, likes \to 0, hasCreator \to 1, replyOf \to 1\}$.

The result for the incoming adjacency table by applying this hash function is depicted in Table 4.5.

For that purpose we move away from the strict schemaless approach of most NoSQL databases. In most applications some kind of data model is used. For our work we assume that a data model is given. This is also the case for the LDBC-SNB and the excerpt in accordance to Figure 4.3 is depicted in Figure 4.1. By considering this data model, we can verify our previous observation. We can store edges of the types likes and knows in the same column of the incoming adjacency table. □

In contrast to the approach of Bornea et al. [Bor+13], we do not use data instances to compute the hash function, but work on data model level. Therefore, we can initialize the database schema and compute hash functions, before we receive any data instances. We can also be sure to have knowledge of all types of edges, and hence can provide robust hash functions that will not create any unforeseeable conflicts. In contrast to this, [Bor+13] rely on using data instances to compute the hash functions. They first need to choose a suitable subset of

| VID | $EID_1$ | $Label_1$ | $SID_1$ | ... | $EID_p$ | $Label_p$ | $SID_p$ |
|---|---|---|---|---|---|---|---|
| **1** | **null** | **null** | **null** | | **[11]** | **hasCreator** | **13** |
| **2** | **[4]** | **knows** | **[1]** | | **null** | **null** | **null** |
| **3** | **[5]** | **knows** | **[1]** | | **null** | **null** | **null** |
| **7** | **[12, 6]** | **likes** | **[1,2]** | | **[14]** | **replyOf** | **[13]** |

Table 4.5.: The desired incoming adjacency table belonging to the graph depicted in Figure 4.3.

the data (since all the data usually is too huge), which is a hard problem on its own. If the subset is not chosen correctly or the data is not yet complete and missing types of vertices or edges types, those types cannot be considered while computing the hash function. Those new types have to be dealt with during runtime of the system, which can be time-consuming and make reorganization of the data necessary.

Our approach does not rely on the suitable choice of a subset of the data, but can be applied without any data instances at all. If we do not have a data model at our disposal, we can fall back on the approach presented by Bornea et al [Bor+13].

Therefore, we can summarize: The goal is to find a hash function based on a given data model that generates a small amount of conflicts (at best none), but at the same time overloads columns whenever two labels do not co-occur at the same type of vertex. Our approach to achieve this goal is described in the next section.

### 4.1.1. Computing Conflict-Free Hash Functions for a Given Data Model

We regard the data model as a graph and generate an interference graph from the data model graph (as proposed by Bornea et al. [Bor+13]), with the difference that we do not rely on choosing a suitable subset of a data instance, but work on the data model itself.

The *input* of the algorithm is the *data model* of the graph that we want to store. The *output* of the algorithm is the *number of columns* we require to provide a collision-free hashing function and the aforementioned *hash function*. This way we provide a mapping that assigns each label (that can occur at an edge of the graph) to a column triple of the adjacency lists. Hence, we construct a hash function for the outgoing and incoming adjacency tables, respectively. To this end we use a graph coloring algorithm on the graph we constructed from the data model.

**Algorithm 4.6:** Algorithm to generate the incoming hash function $\delta_{in}$ from a class diagram.

**Input:** The set of classes C, the function $\rho_C$ that assigns to each class its set of associations.

```
/* Step 1: Convert the data model into the schema graph
   G_schema = (V_schema, E_schema).                          */
```
$V_{schema} \leftarrow \emptyset$;
$E_{schema} \leftarrow \emptyset$;
```
/* Create a node with label L(c) for each class c in the data
   model.                                                    */
```
**foreach** $c \in C$ **do**
  $V_{schema} \leftarrow V_{schema} \cup \{L(c)\}$;
**end**
```
/* Create an edge with Label L(a) for each association a in the
   data model.                                               */
```
**foreach** $c \in C$ **do**
  **foreach** $a \in \rho_C(c)$ **do**
    $E_{schema} \leftarrow E_{schema} \cup \{(sourc(a), targt(a), L(a))\}$;
  **end**
**end**
```
/* Step 2: Convert the schema graph G_schema into the incoming
   interference graph G_in = (V_in, E_in).                   */
```
$V_{in} \leftarrow \emptyset$;
$E_{in} \leftarrow \emptyset$;
$G_{in} \leftarrow \{(V_{in}, E_{in})\}$;
```
/* Create a node for each type of edge in the schema graph.  */
```
**foreach** $e \in E_{schema}$ **do**
  $V_{in} \leftarrow V_{in} \cup \{\lambda_{schema}(e)\}$;
**end**
```
/* Create an edge in the interference graph, if there exists an
   edge in the schema graph with the same target vertex.     */
```
**foreach** $v_1 \in E_{schema}$ **do**
  **foreach** $v_2 \in E_{schema}$ **do**
    **if** $v_1 \neq v_2 \wedge target(v_1) = target(v_2)$ **then**
      $E_{in} \leftarrow E_{in} \cup \{(v_1, v_2)\}$;
    **end**
  **end**
**end**
```
/* As step 3 the outgoing interference graph can analogously be
   created.                                                  */
/* Step 4: Finally, apply a coloring heuristic to get a hash
   function.                                                 */
```
$\delta_{in} \leftarrow Dsatur(G_{in})$;

**Result:** The hash function $\delta_{in}$.

To this end we apply Algorithm 4.6. For the sake of brevity, the algorithm only depicts the generation of the incoming hash function $\delta_{in}$. The computation of the outgoing hash function $\delta_{out}$ can be performed analogously. The overall process is split into four steps:

1. As a preprocessing step, we construct a vertex and edge labeled directed graph $G_{schema} = (V_{schema}, E_{schema})$ representing the data model, where $E_{schema} \subseteq V_{schema} \times V_{schema}$. Associated with each $e \in E_{schema}$ and $v \in V_{schema}$ is a label $L(e)$ and $L(v)$.

   For every class in the data model, we create a vertex to represent the class and assign the class name as its label. For every association, we insert edges for every occurrence this type of association can have (this includes inherited associations) and assign the association's name as label.

2. We construct the undirected interference graph $G_{in} = (V_{in}, E_{in})$ for the hash function of the incoming adjacency table. For every type of edge in the schema graph $G_{schema}$ we want to create a vertex representing the type. Therefore, for every edge $e \in E_{schema}$ from $G_{schema}$ we construct a node that represents the type of edge and add it to the interference graph, if it is not yet present.

   Formally this means:

   - $G_{in} = (V_{in}, E_{in})$;
   - with $V_{in} = \{L(e) | e \in E_{schema}\}$;
   - and $E_{in} = \{((L(e_1), L(e_2)) | \exists e_1, e_2 \in E_{schema}$, such that $e, k_1, k_2 \in V_{schema} \wedge e_1 = (k_1, e), e_2 = (k_2, e) \wedge L(e_1) \neq L(e_2)\}$.

   Note that we allow $k_1 = k_2$ for $E_{in}$. We do this, because if there exist edges of different types between the same two vertices, we want those edges to be mapped to different columns.

3. Analogously, we construct the undirected interference graph $G_{out} = (V_{out}, E_{out})$ for the hashing function for the outgoing adjacency table from $G_{schema}$. Formally:

   - $G_{out} = (V_{out}, E_{out})$;
   - with $V_{out} = \{L(e) | e \in E_{schema}\}$;
   - and $E_{out} = \{((L(e_1), L(e_2)) | \exists e_1, e_2 \in E_{schema}$ such that $e, k_1, k_2 \in V_{schema} \wedge e_1 = (e, k_1), e_2 = (e, k_2) \wedge L(e_1) \neq L(e_2)\}$.

4. We apply a graph coloring heuristic (e.g., Dsatur by Brélaz [Bré79]) on both interference graphs $G_{out}$ and $G_{in}$ to obtain the hash functions $\delta_{out}$ and $\delta_{in}$. We use the definition of graph coloring given by Bornea et al. [Bor+13]. A graph coloring is defined by the interference graph $G_{out}$ and a set of colors $C$. Each edge $e \in E$ denotes a pair of nodes in $V_{out}$ that must be given different colors. A *coloring* is a mapping that assigns each vertex $v \in V_{out}$ to a color $c \in C$. The graph coloring for $G_{in}$ is defined analogously.

Since we construct $G_{out}$ and $G_{in}$ from the underlying data model, we consider any possible type of edge and therefore compute hash functions (and the number of required colors/columns respectively) that do not produce any conflicts. We acquire hash functions that assign each label to a triple of columns by arbitrarily assigning each color to a different triple of columns.

**Remark 4.1**
This approach also enables us to derive a suitable number of columns for a given data model in such a way that we can utilize hash functions that do not produce any conflicts. We do this by supplying the graph coloring algorithm an amount of colors that is higher than necessary. We chose a heuristic that tries to minimize the number of colors in use. Therefore, the number of colors in use may not be optimal. In our experience, this does not negatively influence the efficiency of the approach.

The number of colors the algorithm used to color the graph is the number of column triples required to store the corresponding adjacency list without generating any conflicts.

At the same time, the algorithm constructs the mapping. If the number of required columns is applicable as the number of columns, all neighboring vertices of a given vertex can be retrieved by the database system by loading a single tuple of the according adjacency table. □

**Example 4.7**
We now apply the presented algorithm to (for the sake of brevity) a simplified excerpt of the underlying data model for the LDBC-SNB. This excerpt is depicted in Figure 4.8 as an UML class diagram. The data contains Messages that can be specified further in Posts and

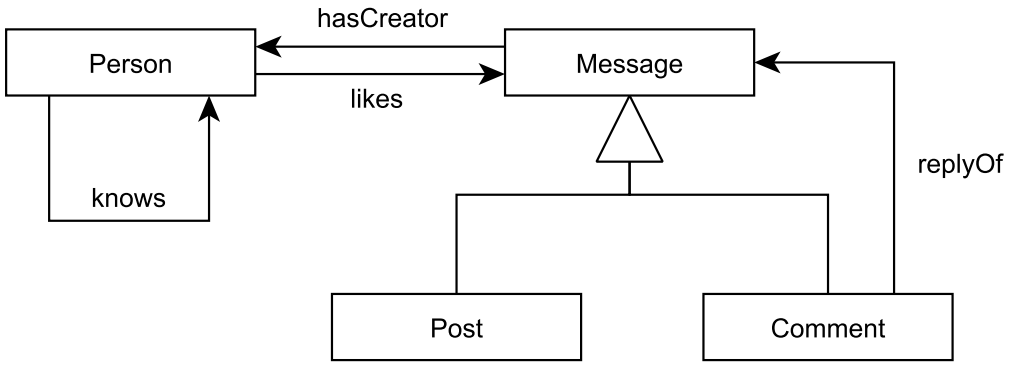



Figure 4.8.: LDBC-SNB data model excerpt

Comments. Both types of Messages can have creators that are Persons. Persons can like both types of Messages. Only Comments can be replies of Messages and a Person can know other Persons.

We follow the four steps previously described:

1. First, we create a vertex for every class in the data model to represent the class and assign the class name as its label. For every association we insert edges for every occurrence this type of association can have (this includes inherited associations) into the graph and assign the association's name as label.

   This results in $G_{schema} = (V_{schema}, E_{schema}, \rho_{schema}, \lambda_{schema})$:

   - $V_{schema} = \{Person, Message, Post, Comment\}$
   - $E_{schema} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$
   - $\rho_{schema} = \{$
     $e_1 \to (Person, Person), e_2 \to (Person, Comment), e_3 \to (Person, Post),$
     $e_4 \to (Person, Message), e_5 \to (Message, Person), e_6 \to (Post, Person),$
     $e_7 \to (Comment, Comment), e_8 \to (Comment, Person), e_9 \to (Comment, Post),$
     $e_{10} \to (Comment, Message)$
     $\}$
   - $\lambda_{schema} = \{$
     $e_1 \to knows, e_2 \to likes, e_3 \to likes, e_4 \to likes, e_5 \to hasCreator,$
     $e_6 \to hasCreator, e_7 \to replyOf, e_8 \to hasCreator, e_9 \to replyOf, e_{10} \to replyOf$
     $\}$

   The result is also depicted in Figure 4.9. Both Post and Comment receive incoming edges with the label likes that were not explicitly given in the original diagram.

2. We compute the incoming interference graph. For every edge $e \in E_{schema}$ from $G_{schema}$ we construct a node in the interference graph. We construct an edge between two vertices $v_1, v_2 \in V_{out}$, if and only if there exist two edges in the schema graph that have the same target vertex and different labels.

   The resulting incoming interference graph $G_{in}$ we constructed from the graph in Figure 4.9 is depicted in Figure 4.10. Since the label types likes and replyOf occur at the same vertex, the corresponding nodes in the interference graph are connected. Because hasCreator has no common target vertices with edges of the types likes and replyOf, no connection between the nodes representing hasCreator and the nodes representing likes and replyOf is included. But it has a common target vertex Person with the edge type knows, therefore the nodes representing hasCreator and knows are connected.

   The formal representation of the undirected graph is $G_{in} = (V_{in}, E_{in}, \rho_{in})$ with

   - $V_{in} = \{knows, hasCreator, likes, replyOf\}$;
   - $E_{in} = \{e_1, e_2\}$;
   - and $\rho_{in} = \{e_1 \to (knows, hasCreator), e_2 \to (likes, replyOf)\}$.

3. The generation of the outgoing interference graph is done analogously to the generation of the outgoing interference graph.

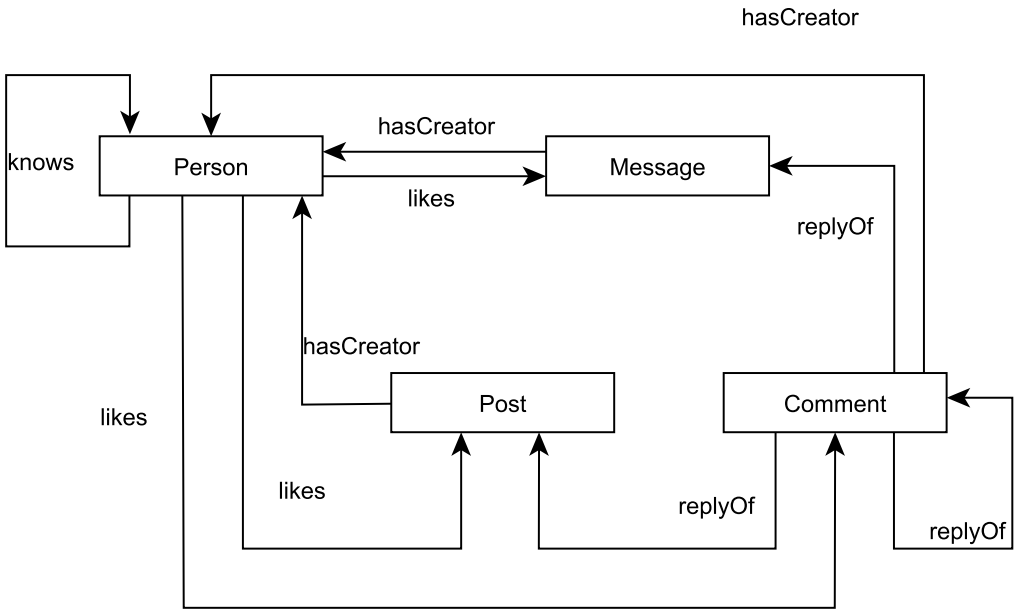



Figure 4.9.: Schema graph constructed from the data model.

4. Finally, we apply a graph coloring heuristic to both interference graphs. An example for the coloring acquired from Figure 4.10 is depicted in Figure 4.11. Since knows and hasCreator are connected, different colors are applied. hasCreator is not connected to replyOf and therefore the same color can be used for both vertices. This means that those two types of edges can be stored in the same columns of the incoming adjacency table. By assigning green to column 0 and blue to 1 we obtain:

   $\delta_{in} = \{knows \to 0, likes \to 0, hasCreator \to 1, replyOf \to 1\}$.

□

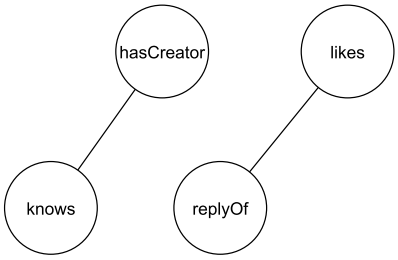



Figure 4.10.: Incoming interference graph generated from the data model in Figure 4.8.

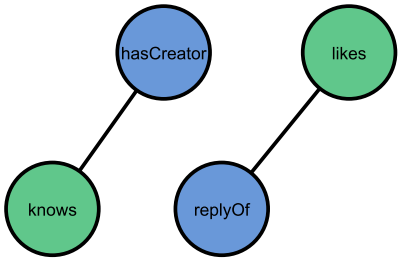



Figure 4.11.: Graph coloring generated from the (partial) interference graph depicted in Figure 4.10.

**Example 4.12**
Let us consider the example graph depicted in Figure 4.3 and the resulting hash function $\delta_{in} = \{knows \to 0, likes \to 0, hasCreator \to 1, replyOf \to 1\}$ of Example 4.7 again.

By considering the range of $\delta_{in}$ defined in Definition 4.2, we see that for $l \in L : 0 \leq \delta_{in}(l) < 2$. Therefore, we only need two column triples to store the edges of the example graph in the incoming adjacency table.

Let us now take a closer look at the different edges of the example graph. The vertex with $VID = 7$ has two incoming edges of the type likes and one incoming edge of the type replyOf. We just computed the incoming hash function $\delta_{in}$, which maps $\delta_{in}(likes) = 0$ and $\delta_{in}(replyOf) = 1$. Hence, we place edges of the type likes and replyOf in different columns.

On the other hand, the nodes with $VID = 2$ and $VID = 3$ have one incoming edge of the type knows with $\delta_{in}(knows) = 0$ and the vertex with $VID = 1$ has an incoming edge of the type hasCreator with $\delta_{in}(hasCreator) = 1$.

| VID | $EID_0$ | $Label_0$ | $SID_0$ | $EID_1$ | $Label_1$ | $SID_1$ |
|---|---|---|---|---|---|---|
| **1** | **null** | **null** | **null** | **[11]** | **hasCreator** | **13** |
| **2** | **[4]** | **knows** | **[1]** | **null** | **null** | **null** |
| **3** | **[5]** | **knows** | **[1]** | **null** | **null** | **null** |
| **7** | **[12, 6]** | **likes** | **[1,2]** | **[14]** | **replyOf** | **[13]** |

Table 4.13.: The resulting incoming adjacency entries by applying the hash function $\delta_{in}$.

At first glance this looks like edges of the types likes and knows could potentially produce conflicts, but since our algorithm is based on the data model, these types of edges never occur at the same vertex as incoming edges. The adjacency table resulting from applying $\delta_{in}$ is depicted in Table 4.13.

□

By applying this process to the **complete** LDBC-SNB data model we have constructed a hashing function for the outgoing adjacency that requires six column triples and a hash

function for the incoming adjacency table that requires four column triples [Sch19b].

## 4.2. Import Strategy

We redundantly store edges such that every edge is stored in the edge table, the incoming adjacency table and the outgoing adjacency table. Since we have to deal with huge amounts of data, we cannot keep the input data in main memory. Therefore, we need to store intermediate data in permanent storage. Accessing this storage is an expensive operation, and therefore we should keep the amount of required accesses low. At the same time, we need to know all the incoming (respectively outgoing) edges of a vertex, before we can create the associated INSERT statement. Otherwise, we would need to update the same statement many times while it has possibly already been stored to disk, resulting in an overhead of disk accesses.

In order to tackle this challenge, several strategies have to be employed. First, we have to make the decision if we want to import data on the client side or on the server side. Since we have to handle data with at least several million nodes and edges, we have to minimize round-trip times. Durand et. al analyzed the efficiency of server-side data loading in comparison to client-side data loading in [Dur+18]. They found that server-side loading usually is orders of magnitude faster than client-side loading. The reasons for this are clear. Server-side loading avoids millions of round-trip times for huge data sets. Therefore, we completely omit the network as a bottleneck and only consider server-side loading for the rest of this work.

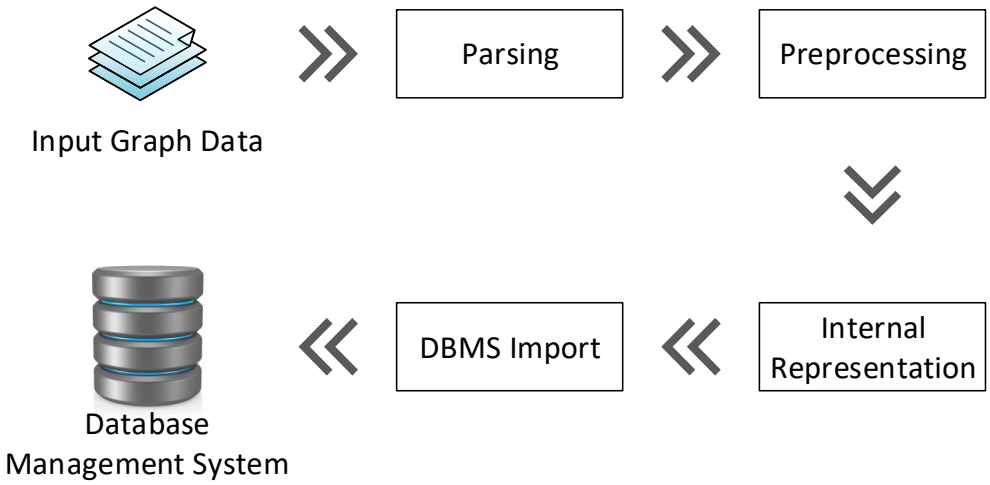



Figure 4.14.: Phases of the bulk data import.

We use the same general approach as [The+16] and [Dur+18] by dividing the import process into several self-contained phases. Since there is no standardized and generally accepted format for the exchange of graph data, we cannot make many assumptions regarding the input data. In particular, we cannot assume that we have a globally unique identifier for every node and edge, which is essential for our storage approach.

We only assume that the following characteristics are given:

- The data can be transformed to a property graph that complies with Definition 3.1 of Section 3.1;
- and the target database does not yet contain data.

In particular, we do **not** assume that

- the data is given in the property graph format;
- a unique identifier for every node and edge exists;
- or we receive a specific number of files or can rely on a specific file structure.

We now describe the running example that we use for this chapter.

**Example 4.15**
We will use the following data set (a simplified LDBC-SNB instance) as input files for a running example in this chapter. It is based on the graph depicted in Figure 3.3.

| **Persons** | | |
|---|---|---|
| id | name | firstName |
| 1 | Silva | Ana |
| 2 | Yamamoto | Akira |
| 3 | Lepland | Carmen |

| **Person knows Person** | | |
|---|---|---|
| source | target | since |
| 1 | 2 | 14.06.2018 |
| 1 | 3 | 21.03.2016 |

| **Person likes Post** | |
|---|---|
| source | target |
| 2 | 2 |
| 1 | 2 |

| **Posts** | |
|---|---|
| id | creationDate |
| 1 | 02.03.2020 |
| 2 | 03.03.2020 |

| **Post has Creator** | |
|---|---|
| source | target |
| 1 | 2 |

| **Post replyOf Post** | |
|---|---|
| source | target |
| 1 | 2 |

| **Person e-Mail** | |
|---|---|
| id | email |
| 1 | silva.ana@gmail.com |
| 1 | ana.silva@outlook.com |

Table 4.16.: Example LDBC-SNB comma-seperated values (CSV) files.

The import of LDBC-SNB data is challenging and contains most of the interesting aspects of data import into our graph schema. Since the data is generated, we can consider different data set sizes. The LDBC-SNB data is provided in three different types of CSV files.

**Vertex files** Vertices are labeled. This specifically means that vertices can be grouped into types. For example, vertices of type Person, Place, or Post exist in the data set. For each type an input file is provided. This file describes the set of vertices, the vertex identifier and its single-valued attributes as a CSV list. An important aspect of the LDBC-SNB data set is the property that identifiers are only unique within the specific vertex type. In particular this means that new identifiers have to be assigned, since our approach requires global identifiers (see Section 3.2).

**Multi-valued attribute files** If an attribute can contain multiple values (in this case e.g., e-mail addresses), these are provided in additional files. For every vertex type and every multi-valued attribute an additional file has to be accessed.

**Edge file** Just as vertices, edges are labeled. For every type of edge a CSV file is provided. In contrast to our approach, edges in the LDBC-SNB data set do not possess an identifier at all. If the edge type has additional information attached, those attributes are also provided in the file. Multi-valued attributes do not exist for edges.

As information about a single vertex can be distributed among different files and a globally unique identifier is not provided, we have to include a preprocessing step. As we can see, the identifiers 1 and 2 are both used for vertices of type Person and Post, while edges do not have an identifier assigned. Therefore, during the import process we need to make sure that we can assign unique identifiers for vertices as well as edges. □

In the subsequent sections the four phases of the data import (depicted in Figure 4.14) is shown in the following sections: First, how the data can be parsed. Second, how it can be preprocessed. Third, the internal representation and finally, our graph import algorithm that shows how data can be imported in **RATG** as well as **SQLGraph** and therefore makes both approaches usable for arbitrary graph data sets.

### 4.2.1. Parsing the Input Data

The first phase parses the input data. We interpret the data as a property graph according to Definition 3.1. This means that the data does not have to be supplied as a property graph, as long as an applicable mapping into the property graph model can be provided. For example, we can import RDF data as a property graph by defining a suitable mapping. This has been discussed by Hillinger in [Hil20]. Since we do not assume a specific input format, this step is vital to assure the efficient import of the whole graph. In many cases this step and the preprocessing of the data can and should be performed in a single process.

Data parsing is highly dependent on the input data set. For example, the LDBC-SNB provides multi-valued attributes in separate files (see Table 4.16). To this end, we merge the vertices with their attributes in the parsing phase in order to avoid costly update operations later on. Since we perform the parsing and the preprocessing phase in a single step to increase performance, we will show an example in the next section describing the preprocessing.

### 4.2.2. Preprocessing of the Input Data

After the data has been parsed, we perform an optional preprocessing step. In this phase we assure the integrity of constraints. Other tasks that are performed during this phase can include checking of integrity constraints, existence of vertices connected by edges or aggregation tasks [Hil20].

Specifically in the case of the LDBC-SNB, the reassignment of unique identifiers is required. The LDBC-SNB provides a file for every type of vertex included in the data set. But the identifier assigned to the vertices is only unique within the type of the vertex. This phase is also highly dependent on the input data and the interleaved execution with **parsing** can be beneficial for performance.

**Example 4.17**
Considering that the data is available in a CSV format as depicted in Example 4.15, we can use standard technology to parse the data. For this data set we perform three tasks in the preprocessing phase of the LDBC-SNB data import:

1. We sort input files containing edges by the source vertex id. This step is not crucial to be able to import the data, but it simplifies the process to transform the input data to our internal representation. Because we have to merge data from several files to obtain all data regarding a single node, sorted files enable us to transform all data in a single pass for every file. Since data size can easily exceed available main memory, we can use an external sorting algorithm like Knuth describes in [Knu73]. As can be seen in Table 4.16, vertex data is distributed in several files (the vertex data itself in one file and multi-valued attributes in additional files). We use a **Merge sort** approach to merge vertice and attribute data.
2. For each vertex we assign a globally unique identification number.
3. We assign a globally unique identifier to each edge of the input data.

□

After preprocessing, we are ready to convert the input data into our internal property graph model.

### 4.2.3. Converting the Data into the Internal Representation

We create an internal property graph representation of the input data. After the parsing and preprocessing has been finished, we create an internal representation of the property graph that is independent of the input data format. This model represents a graph according to Definition 3.1. In particular, at this point we can assume that we have a unique identifier for every vertex and edge. Once we have the data available in our internal representation, the process is completely independent of the input format. Among other things, this means that we can store this internal representation and restart the process from this point on for future

loading. One should take note that this representation provides a very simple graph format and is not intended for other uses than intermediate storage. From this point on, the import process is independent of the input data format.

**Example 4.18**
The input data is transformed into an internal graph representation. Examples of the data after preprocessing and conversion to the internal representation are depicted in Tables 4.19 and 4.20

| **Vertices** | |
|---|---|
| VID | Attributes |
| 1 | type: Person; name: Yamamoto; firstName: Akira |
| 2 | type: Person; name: Silva; firstName: Ana |
| 3 | type: Person, name: Lepland; firstName: Carmen |
| 7 | type: Post, creationDate: 03.03.2020 |
| 13 | type: Post, creationDate: 02.03.2020 |

Table 4.19.: Vertices after preprocessing and conversion into the internal representation.

| **Edges** | | | | |
|---|---|---|---|---|
| EID | SID | TID | Label | Attributes |
| 4 | 1 | 2 | knows | since: 14.06.2018 |
| 5 | 1 | 3 | knows | since: 21.03.2016 |
| 6 | 2 | 7 | likes | |
| 11 | 13 | 1 | hasCreator | |
| 12 | 1 | 7 | likes | |
| 14 | 13 | 7 | replyOf | |

Table 4.20.: Edges after preprocessing and conversion into the internal representation.

As described in Section 1.1, our work is partly motivated by the storage of huge amounts of data that do not fit in main memory. Therefore, we applied an outsourcing technique to store the internal representation on hard drive. Particularly this is necessary to import the LDBC-SNB data sets, since generating the data using higher scale factors creates up to several hundred gigabytes of data. The number of vertices or edges that can be held in-memory varies greatly from machine to machine. To this end, we performed a short evaluation to determine the influence of the number of elements that are held within memory on the run-time of the import process, since writing to disk too often has a negative influence on import efficiency. A short evaluation analyzing the influence of the number of internally stored elements can be found in Section 4.3. □

### 4.2.4. Writing the Data into the DBMS

Finally, the data is imported into the database. The algorithm presented in this section can be used to import data into **RATG**, as well as **SQLGraph**, and therefore lets us use both approaches on arbitrary data sets that satisfy the assumptions we presented in Section 4.2.

The data is read from the internal representation and converted into the desired format. From here on, the process is once again depending on the target database system. Different strategies for writing the data into the database can be applied. We propose to use the bulk loading mechanisms that most current database systems offer. Since these mechanisms are specifically designed to load huge amounts of data in the most efficient manner, we leverage this advantage. We create files suited for the specific database system loading mechanism from our internal representation. Since there is no standardized bulk loading mechanism, even in the context of RDBMS, there is no way to design this phase independent of the database system.

This is the last task that could potentially be performed on the client machine. If the process was carried out on the client, we now need to transfer those files to the database server machine. We can now use the bulk loading mechanism to populate the database.

**Algorithm 4.21:** Steps of the database import.

**Input:** A property graph $G = (N, E, \rho, \lambda, \sigma)$ and the hash functions $\delta_{out}, \delta_{in}$
dropIndices();
createVertices(N, $\sigma$);
createOutgoingAdjacency(E, $\rho$, $\lambda$, $\sigma$, $\delta_{out}$);
createIncomingAdjacency(E, $\rho$, $\lambda$, $\sigma$, $\delta_{in}$);
createIndices();
**Result:** A database instance containing $G$.

For our algorithm, we assume the following properties:

- The property graph data we need to import into the database is huge, and we are **not able to fit the data into main memory**. This means that our algorithm has to store intermediate results (e.g. insert statements) on hard drive. We also have to load the data we want to import in chunks. If the data size of the data set that is to be imported is smaller, we can use the same approach. We then set parameters that handle the externalization of data to values high enough, such that the import is mostly handled in-memory.
- The **resulting database instance** and **intermediate files fit** into the **available disk space**.
- Since we need to load data from disk and store intermediate results on disk, **access time to the permanent storage is a major bottleneck**. Therefore, we need to minimize the number of times the same vertex or edge has to be accessed.
- If too many database commands are sent, **round-trip time is another bottleneck** we need to avoid. This even is the case while running the import program on the

database machine. Therefore, we try to keep the number of messages to the database limited. This means that we want to only import complete vertex, edge and adjacency tuples in order to avoid update operations on existing data. Those insert operations can be performed by means of batch processing or DBMS-dependent bulk-loading mechanisms.

- We already have suitable hash functions available (which can be computed as described in Section 4.1) and we therefore know their range and the **number of columns** required (as described in Remark 4.1).
- The **database schema** has already been **created**.
- Finally, we assume that we can hold the **adjacency for an arbitrary single vertex in main memory**. If this is not the case, we can adjust our import algorithm by primarily sorting by source vertex and secondarily sorting by edge label. We sort labels by the order in which they are assigned to columns. We can then process one label at a time and write the corresponding import files to disk. Although, we assume that once a graph reaches the size that the neighborhood of a single vertex does not fit into main memory, the overall performance of any graph database will not be adequate on that particular hardware configuration.

**Algorithm 4.22:** Creation of insert statements for the vertices.

**Input:** The set of vertices N and $\sigma$
$vertices \leftarrow \emptyset$;
**foreach** $(n \in N)$ **do**
  $vertices \leftarrow vertices \cup createInsertStatement(n, \sigma)$;
**end**
$importData(vertices)$;
**Result:** A database instance that contains the vertex data.

Note that we use database bulk import functionality instead of *SQL* insert-statements. Bergmüller [Ber19] demonstrates that this is a far more time-efficient import method than using batch inserts. Since the file format for this import varies highly from DBMS to DBMS, we use insert statements for the description of the algorithms in this section. Those statements can be easily replaced with the corresponding bulk import lines required for the desired target database.

The overall process is depicted in Algorithm 4.21 and can be divided into 5 steps:

**1) Dropping existing indices** We assume that we are importing huge amounts of data. Since the constant reorganization of existing indices is a considerable overhead, we drop all indices on the tables. Newly creating the indices again after the data import has finished proved to be a much more efficient strategy.

**2) Importing vertices** Next we import the vertices into the database (see Algorithm 4.22). Because we have already performed cleanup and the assignment of unique identifiers to the vertices in the previous phases, the only task to perform for vertices is the transformation into the syntax compatible to the target DBMS.

**3) Importing edges and creating the outgoing adjacency** Because we want to minimize disk access, we perform the creation of the input format for the edge list and the outgoing adjacency in the same step (see Algorithm 4.24). First, the edges of the graph are sorted by their source vertex using an external sorting algorithm (e.g. see [Knu98]). After sorting, every time we encounter a new vertex identifier we know that we have seen all outgoing edges for the last vertex and we can create the final adjacency entry for this vertex.

Therefore, we gather all edges belonging to the currently handled vertex until we find a new identifier. When we found a new id, we finalize the adjacency of the last vertex by creating the associated adjacency entry.

Algorithm 4.25 depicts the creation of adjacency entries. First, we iterate over all edges of the input vertex and sort them into buckets corresponding to the available column triples in the database schema. Then each bucket is transformed into the corresponding columns for the input statement and appended to the statement.

Note that it can be necessary to temporarily store the statements to disk, because we do not want to create a query for every single tuple in order to avoid round-trip times. The frequency with which data is written to disk is a tuning parameter that depends on the hardware the algorithm is running on.

While we loop over all edges, we create – in analogy to the creation of the vertex statements – the statements for the edge list table. Once we have created all the statements, we start the import to the database. By applying this algorithm, we only have to stream the edge data once from disk in order to create all edge list and outgoing adjacency entries.

**4) Creating incoming adjacency** Analogously to the creation of the outgoing adjacency we can create the incoming adjacency (see Algorithm 4.23). To this end, we sort the set of edges $E$ by target vertex. Afterwards we can create the import statements vertex by vertex as seen in the creation step for the outgoing adjacency.

**5) Creating indices** As the last step of the import process, we create the indices we have previously described in Section 3.2.1.

**Algorithm 4.23:** Import for the incoming adjacency tables.

**Input:** The set of edges E, the functions $\rho$, $\lambda$, $\sigma$ and the hash function $\delta_{in}$.

```
/* The result list file for the insert statements                    */
resultAdjacency ← ∅;
/* A not existing id to initialize the loop                          */
currentVertexId ← MINIMUM_VERTEXID − 1;
/* The current set of edges for a single vertex                      */
adjacency ← ∅;
/* Sort edges ascending by target id in order to create the
   incoming adjacency entries for each vertex.  An external sorting
   algorithm is used.                                                */
E_sorted ← sortByTargetVertex(E, ρ);
foreach e ∈ E_sorted do
    /* Are we still processing the same vertex?                      */
    if (currentVertexId ≥ target(e)) then
        adjacency ← adjacency ∪ {e}
    else
        /* Create entries for the now finished vertex                */
        resultAdjacency ←
          resultAdjacency ∪ createIAStatement(currentVertexId, adjacency, ρ, λ, δ_in);
        /* Start the collection of edges for the next vertex         */
        adjacency ← {e} ;
        currentVertexId ← target(e) ;
    end
end
/* Create statements for last vertex                                 */
resultAdjacency ← resultAdjacency ∪ createIAStatement(adjacency, ρ, λ, δ_out);
/* Write data to the database                                        */
importData(resultAdjacency);
```

**Result:** A database instance that contains the incoming adjacency entries for $G$.

**Algorithm 4.24:** Import for the edge list and outgoing adjacency tables.

**Input:** The set of edges E, the functions $\rho$, $\lambda$ and the hash function $\delta_{out}$.

```
/* The result list files for the insert statements                    */
```
$resultList \leftarrow \emptyset$;

$resultAdjacency \leftarrow \emptyset$;

```
/* A not existing id to initialize the loop                           */
```
$currentVertexId \leftarrow MINIMUM\_VERTEXID - 1$;

```
/* The current set of edges for a single vertex                       */
```
$adjacency \leftarrow \emptyset$;

```
/* Sort edges ascending by source id in order to create the
   outgoing adjacency entries for each vertex.  An external sorting
   algorithm is used.                                                 */
```
$E_{sorted} \leftarrow sortBySourceVertex(E, \rho)$;

**foreach** $e \in E_{sorted}$ **do**

  $resultList \leftarrow resultList \cup createListStatement(e, \rho, \lambda, \sigma)$;

  ```
  /* Are we still processing the same vertex?                         */
  ```

  **if** $(currentVertexId \geq src(e))$ **then**

    $adjacency \leftarrow adjacency \cup \{e\}$

  **else**

    ```
    /* Create entries for the now finished vertex                     */
    ```

    $resultAdjacency \leftarrow$
    $resultAdjacency \cup createOAStatement(currentVertexId, adjacency, \rho, \lambda, \delta_{out})$;

    ```
    /* Start the collection of edges for the next vertex              */
    ```

    $adjacency \leftarrow \{e\}$ ;

    $currentVertexId \leftarrow src(e)$ ;

  **end**

**end**

```
/* Create statements for last vertex                                  */
```
$resultList \leftarrow resultList \cup createListStatement(adjacency, \rho, \lambda, \sigma)$;

$resultAdjacency \leftarrow resultAdjacency \cup createOAStatement(currentVertexId, adjacency, \rho, \lambda, \delta_{out})$;

```
/* Write data to the database                                         */
```
$importData(resultList, resultAdjacency)$;

**Result:** A database instance that contains the edge list and outgoing adjacency entries for $G$.

**Algorithm 4.25:** Creation of incoming adjacency statements.

**Input:** The list of incoming edges $E_v$ for node $v$, the functions $\rho$, $\lambda$, $\sigma$ and the hash function $\delta_{in}$.

```
/* Create an array that can hold all column triples.              */
```
$edgeBuckets \leftarrow Array[max(range(\delta_{in})) + 1]$;
```
/* Organize triples.                                              */
```
**foreach** $(e \in E_v)$ **do**
  $position \leftarrow \delta_{in}(\lambda(e))$;
  $edgeBuckets[position] \leftarrow add(edgeTriples[position], e)$;
**end**
```
/* Initialize the import statement with the vertex id.            */
```
$adjacencyStatement \leftarrow appendSourceId(v)$;
**foreach** $(bucket \in edgeBuckets)$ **do**
  $eIdColumn \leftarrow []$;
  $labelColumn \leftarrow \lambda(bucket.first())$;
  $sIdColumn \leftarrow []$;
  **foreach** $(edge \in bucket)$ **do**
    $eIdColumn \leftarrow appendEdgeId(eIdColumn, edge)$;
    $sIdColumn \leftarrow appendSourceId(tIdColumn, src(edge))$;
  **end**
  $adjacencyStatement \leftarrow append(adjacencyStatement, eIdColumn, labelColumn, tIdColumn)$;
**end**
$adjacencyStatement \leftarrow finish(adjacencyStatement)$;
**return** *adjacencyStatement*;

**Result:** A statement that creates the adjacency entry for vertex $v$.

**Example 4.26**
Since the creation of the INSERT statements for the vertex and edge table is straightforward, we only take a closer look at the generation of the incoming adjacency table (see Algorithm 4.23). Creation of the outgoing adjacency table statements can be done essentially the same way.

Let us assume the list of edges depicted in Table 4.20 provides the input data for the example.

| **Edges** | | | | |
|---|---|---|---|---|
| EID | SID | TID ↑ | Label | Attributes |
| 11 | 13 | 1 | hasCreator | |
| 4 | 1 | 2 | knows | since: 14.06.2018 |
| 5 | 1 | 3 | knows | since: 21.03.2016 |
| 6 | 2 | 7 | likes | |
| 12 | 1 | 7 | likes | |
| 14 | 13 | 7 | replyOf | |

Table 4.27.: Edges sorted ascending by the target id TID.

After initializing the variables, the first step of the algorithm is to sort the set of edges $E$ by the target vertex $target(e)$ with $e \in E$. The result of the sorting algorithm is depicted in Table 4.27.

Then we iterate over the edges in sorted order with the initial values $resultList = \emptyset$, $resultAdjacency = \emptyset$, $currentVertexId = 0$ and $adjacency = \emptyset$:

- $e = e_{11}$:
  Since $currentVertexid = 0$ and $target(e_{11}) = 1$ we move to the *else* branch. Because $adjacency = \emptyset$, no statement is created. Result of this iteration:
  - $resultAdjacency = \emptyset$
  - $adjacency = \{e_{11}\}$
  - $currentVertexId = n_1$
- $e = e_4$:
  Since $currentVertexid = n_1$ and $target(e_4) = n_2$ we skip to the *else* branch. Because $adjacency = \{e_{11}\}$, an import statement $is_1 = is_{currentVertexId}$ is created and added to the list of already created import statements. Result of this iteration:
  - $resultAdjacency = \{is_1\}$
  - $adjacency = \{e_4\}$
  - $currentVertexId = n_2$
- $e = e_5$:
  Since $currentVertexid = n_2$ and $target(e_5) = n_3$ we skip to the *else* branch. Because $adjacency = \{e_4\}$, an import statement $is_{currentVertexId}$ is created and added to the list of already created import statements. Result of this iteration:

  - $resultAdjacency = \{is_1, is_2\}$
  - $adjacency = \{e_5\}$
  - $currentVertexId = n_3$

- $e = e_6$:
  Since $currentVertexid = n_3$ and $target(e_6) = n_7$ we skip to the *else* branch. Because $adjacency = \{e_5\}$, an import statement $is_{currentVertexId}$ is created and added to the list of already created import statements. Result of this iteration:

  - $resultAdjacency = \{is_1, is_2, is_3\}$
  - $adjacency = \{e_6\}$
  - $currentVertexId = n_7$

- $e = e_{12}$:
  Since $currentVertexid = n_7$ and $target(e_{12}) = n_7$ with $n_7 \geq n_7$, we add the current edge to the incoming adjacency set for $n_7$. Result of this iteration:

  - $resultAdjacency = \{is_1, is_2, is_3\}$
  - $adjacency = \{e_6, e_{12}\}$
  - $currentVertexId = n_7$

- $e = e_{14}$:
  Since $currentVertexid = n_7$ and $target(e_{12}) = n_7$ with $n_7 \geq n_7$, we add the current edge to the incoming adjacency set for $n_7$. Result of this iteration:

  - $resultAdjacency = \{is_1, is_2, is_3\}$
  - $adjacency = \{e_6, e_{12}, e_{14}\}$
  - $currentVertexId = n_7$

After we have iterated over all edges, the last processed vertex is finished and the INSERT statement $is_7$ for node $n_7$ is added to the list of adjacency statements resulting in $resultAdjacency = \{is_1, is_2, is_3, is_7\}$. We take a closer look at the creation of the import statement for $n_7$ in the following example. The list of import statements can now be transferred to the database and executed. □

**Example 4.28**
As mentioned in Example 4.26 we now take a closer look at the creation of a single INSERT statement for the incoming adjacency table depicted in Algorithm 4.25.

Let's assume $\delta_{in} = \{likes \rightarrow 0, hasCreator \rightarrow 0, knows \rightarrow 1, replyOf \rightarrow 1\}$ in accordance with the hash function generated in Example 4.7. Also let the current vertex be $n_7$ with $E_v = \{e_6, e_{12}, e_{14}\}$. We initialize $edgeBuckets = Array[max(range(\delta_{in})) + 1] = Array[2] = [\emptyset, \emptyset]$.

Then we iterate over the adjacent edges for the current vertex and sort them into the buckets, resulting in: $edgeBuckets = [\{e_6, e_{12}\}, \{e_{14}\}]$.

The first part of the INSERT statement, which is not depending on the edges of the v vertex, is created as
$adjacencyStatement =$ "$INSERT\ INTO\ IncomingAdjacency\ VALUES\ [...](7$".

Afterwards, we sequentially append each bucket:

- $bucket = \{e_6, e_{12}\}$:
  We initialize $eIdColumn = []$, $labelColumn = likes$ and $sIdColumn = []$.

  Now, we iterate over the edges gathered in the bucket and gather all edge ids and source vertex ids in order resulting in:

  - $eIdColumn = [6, 12]$
  - $sIdColumn = [2, 1]$

  Before proceeding to the next bucket we can append those columns to the INSERT statement:
  $adjacencyStatement =$ "INSERT INTO [...] VALUES (7, [6, 12], likes, [2, 1]".

- $bucket = \{e_{14}\}$:
  We initialize $eIdColumn = []$, $labelColumn = replyOf$ and $sIdColumn = []$. Gathering the edges in the bucket results in:

  - $eIdColumn = [14]$
  - $sIdColumn = [13]$

  We append the collected column to the statement:
  $adjacencyStatement =$ "INSERT INTO [...] VALUES (7, [6, 12], likes, [2, 1], [14], replyOf, [13]".

After all buckets have been appended to the statement, the statement is finalized and returned with the result:
"INSERT INTO [...] VALUES (7, [6, 12], likes, [2, 1], [14], replyOf, [13])".

□

In the course of this work we applied this import concept on the LDBC-SNB data set, RDF (see [Hil20]) and our IFC use case (see Chapter 7) in various forms to store the data in our database schema.

**Remark 4.2**
As we have stated above, we do not use batches of INSERT statements to achieve the best data import performance. The most efficient way to import data is using DBMS internal functions like the **PostgreSQL** COPY mechanism, which is a CSV import mechanism. The presented import algorithm can easily be adapted by writing the created insert statements to a CSV file that can then be used as input for the DBMS file import mechanism. □

## 4.3. Import Algorithm Performance Evaluation

To show the applicability of our approach described in Section 4.2, we performed an evaluation using the LDBC-SNB data set described in Section 2.4. This shows that our approach to import data works for arbitrarily large data sets, as long as the assumptions presented in Section 4.2.4 hold.

In addition, all examples in the previous sections were based on this data set. The following evaluations were performed on a dedicated server with two Intel Xeon 2.6GHz CPUs (in total 8 cores), 96GB memory and a 6 SSD RAID-0 running 64-bit Linux.

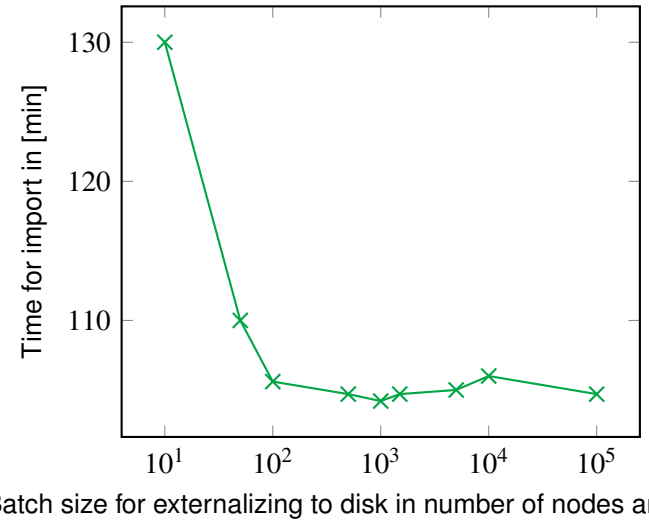



Figure 4.29.: Time required to perform a complete data import for LDBC-SNB data set (scale factor 30) in regard to the configured batch size.

As described above, intermediate results for the insert statements are periodically written to disk. We conducted a short preliminary evaluation to analyze the behavior of import times in regard to the size of batches written to disk. We expected to find that if we increase batch size, performance also increases up to a certain threshold. Once we reach this batch size, the import performance remains the same. At this point we have minimized the overhead of disk access while we can already start preparing the next batch that will be written to disk. For data sets that are small enough, choosing a very high batch size transforms this import mechanism into an in-memory approach.

The performance in regard to the batch size is depicted in Figure 4.29. As we can see, choosing a small number (like internally processing ten vertices or edges) leads to an overhead by writing to disk too often. Increasing the number of vertices kept in memory also increased the performance of the import mechanism. At around $1,000$ internally processed elements we reach the best performance for this hardware configuration. Additional increases in used memory do not shorten the time required to import data, since parallelism in writing data

| SF | Approx. #Vertices | Approx. #Edges |
|---|---|---|
| **1** | 3,200,000 | 17,300,000 |
| **3** | 9,300,000 | 52,700,000 |
| **10** | 30,000,000 | 176,600,000 |
| **30** | 99,400,000 | 655,400,000 |
| **100** | 317,700,000 | 2,154,900,000 |
| **300** | 907,600,000 | 6,292,500,000 |

Table 4.30.: Approximate number of vertices and edges of different LDBC-SNB scale factors.

to disk and preparing the next batch to write cannot be used further to our advantage. One should note that the optimal value depends on the hardware configuration used for the data import. We used this preliminary evaluation to find a good configuration for our available hardware for further evaluations.

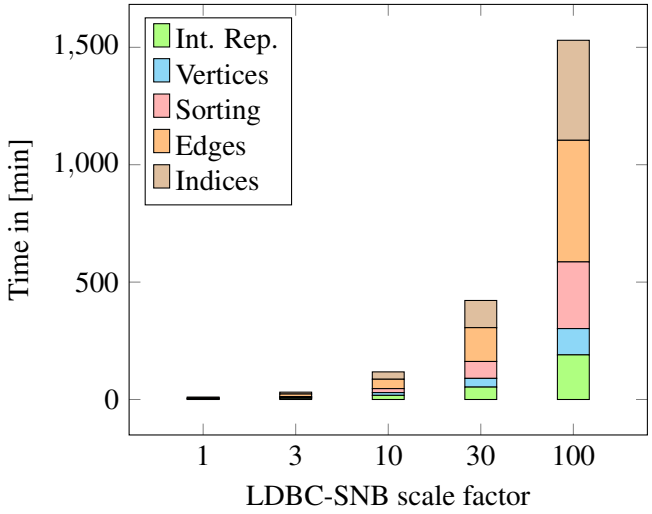



Figure 4.31.: Required time of the different import phases depending on the LDBC-SNB scale factor.

Afterwards, we performed a performance evaluation of our data import algorithm. We imported the data set for scale factors 1, 3, 10, 30, and 100 into an empty database. Each import was performed at least five times and the mean duration is discussed here, since the required time did not vary significantly . An overview of the number of vertices and edges of the different scale factors is depicted in Table 4.30.

### 4.3.1. Runtime Complexity Analysis and Evaluation

We assume that the hash function computation has been performed as a preliminary step. Therefore, we expect the algorithm to have a runtime behavior in $\mathcal{O}(m\log(m))$ with $m = |E|$ where we assume $|N| \leq |E|$.

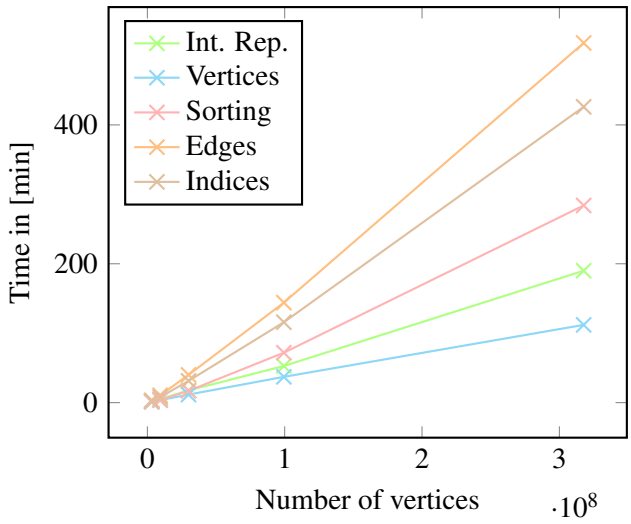



Figure 4.32.: Required time for the data import depending on the number of vertices.

Step 1 (dropping indices, if created beforehand) can be performed in constant time and is handled by the DBMS. Creating the internal representation (see Algorithm 4.22) is a linear routine processing each node and edge once. Afterwards the edge list needs to be sorted twice: First, to create the outgoing adjacency table (see Algorithm 4.24) and second to create the incoming adjacency table (see Algorithm 4.23). Since we sorted the edges accordingly, we can create the adjacency lists for each vertex in one step and only need to handle each edge once. Therefore, this computation step is dominated by the sorting algorithm and is well known to require $\mathcal{O}(m\log(m))$ with $m$ being the number of edges. Step 5 (creating the indices again) is handled by the DBMS and usually requires $\mathcal{O}(n\log(n))$ (applying the worst case by inserting all elements into a B-Tree [Com79]). Since these steps are performed sequentially, the overall runtime is dominated by the sorting algorithm and requires $\mathcal{O}(m\log(m))$ with $m$ being the number of edges.

Figure 4.31 depicts the time required to import different scale factors, split into the different import phases as described above in Section 4.2. Creating the edge list, the outgoing adjacency and the incoming adjacency tables are aggregated into a single phase, since the complete execution is necessary to properly leverage the advantages of **RATG**. As we can see the overall time that is required to import the data grows nearly linearly in regard to the scale factor.

Yet, the scale factor alone is not a sound parameter in regard to our previous runtime analysis. To get a better insight of the behavior of the different import phases consider Figure 4.32

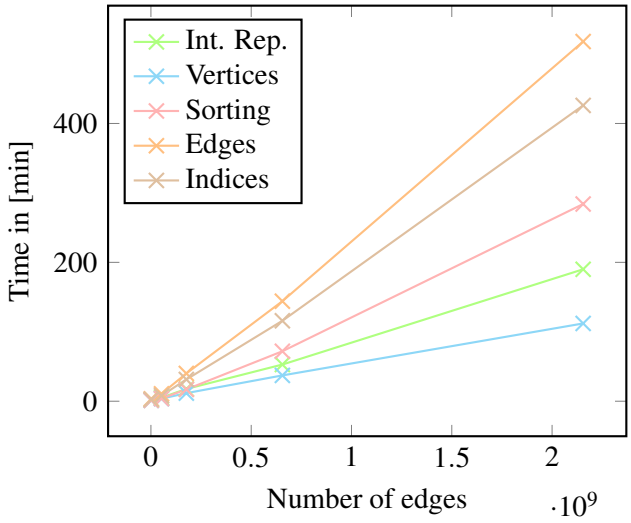



Figure 4.33.: Required time for the data import depending on the number of edges.

and Figure 4.33, which depict the required import time in regard to the number of vertices and edges of the imported data graph. As it was expected, we can see the time required to generate the import statements for vertices and edges grows linearly in the number of vertices and edges respectively. The time for sorting the edges grows surprisingly slowly and only slightly faster than linear, even though the theoretical runtime is in $\mathcal{O}(m \log(m))$. The overall import performance is in $\mathcal{O}(m \log(m))$.

# 5. Data Retrieval

One of the main applications for database systems has always been data retrieval. In this chapter, we describe the different ways data can be retrieved from our property graph storage schema. We first describe how to retrieve data using *SQL*. Since even simple graph pattern matching *SQL* queries for the proposed schema can get very complicated and lengthy, we introduce a way to integrate the *Cypher* query language into our system. *Cypher* is a dedicated property graph query language that is intuitive and powerful. Through the integration of *Cypher* we achieve an easily applicable approach.

## 5.1. Basic Graph Queries and Operations

To retrieve data from **RATG** in an effective way, we need to be able to fetch data about nodes and edges that connect those nodes. The database schema is depicted in Table 3.4. Since we have three different tables that store edge data, we can use different options to query neighborhoods of vertices.

**Retrieving Vertices** All vertex data is stored in the vertex table. An example for the table is depicted in Table 3.5. We can retrieve the data using standard *SQL* by e.g. accessing the vertex's VID, or we can receive vertices by leveraging the inverted index we create and use JSON query functionality.

**Example 5.1**
An example of how to search for a vertex with a specific name and return the first name and name of the vertex (in **PostgreSQL** dialect) is depicted in Listing 5.2.

```
SELECT attributes->>'firstName', attributes->>'name'
FROM vertices
WHERE attributes @> '{"name":"<?>"}'
```

Listing 5.2: Example neighbourhood query for a single node using the edge list table.

□

**Retrieving the Neighborhood of a Vertex** Whenever we want to retrieve the (outgoing) neighborhood of a query we have to make a choice: Either we use the edge list (an example is depicted in Table 3.6) or the adjacency representation. As depicted in Listing 5.4, we can easily query the neighborhood of a vertex using the edge list table and a standard *SQL* query. We can also search for edges by using the attribute's column the same way as shown in Listing 5.2 for vertices.

**Example 5.3**
The example depicted in Listing 5.4 shows, how to retrieve the source vertex id, the lable and the target vertex id of an edge.

```
SELECT SID, LABEL, TID
FROM edges
WHERE SID = <?> AND label = <?>
```

Listing 5.4: Example neighbourhood query for a single node using the edge list table.

□

**Remark 5.1**
Any time we need to access the attributes of edges, we have to use the edge list, since the adjacency lists do not store edge attributes. □

**Using the Adjacency Lists for Neighborhood Queries** Whenever it's not needed to consider the attributes of an edge while retrieving the neighboring vertices, we can use the adjacency lists (examples depicted in Tables 3.8 and 3.9). Our evaluations have shown that for queries that retrieve paths of fixed length, it is more efficient to use the adjacency lists than using the edge list. For a detailed description of our graph store evaluation see Chapter 6.

Even though we need to have a mapping from edge types to columns, a very useful property of the previously described schema (see Section 3.2) is the ability to query the adjacency of a vertex without requiring knowledge about the applied hash function. We can achieve this by using array functionality that most up-to-date database systems support. We created a prototypical implementation for both PostgreSQL and OracleDB [Kid20].

**Example 5.6**
An example of the general query structure implemented in the **PostgreSQL** dialect is depicted in Listing 5.5: The first CTE *unshred_edges* converts the list of tuples belonging to one vertex, of which every row contains $k$ edge types, into a list of rows that contains one edge type per row. The second CTE *gather_edges* then splits those tuples into a list of edges that represents a well-known edge structure that is the same as the result of the query depicted in Listing 5.4. Analogously, we can query the incoming neighborhood of a vertex (see Listing 5.7). □

```
WITH unshred_edges AS (
      SELECT vertexid AS sourceid,
       UNNEST(array[label_0, ..., label_k]) AS label,
       UNNEST(array[target_0, ..., target_k]) AS tmp
      FROM OutgoingAdjacency
      WHERE vertexid = <?>
),
gather_edges AS (
      SELECT sourceid, label, array_elements(tmp) AS targetid
      FROM unshred_edges
      WHERE label = <?>
)
SELECT sourceid, label, targetid
FROM gather_edges
```

Listing 5.5: Example outgoing neighbourhood query for a single node using the outgoing adjacency table.

```
WITH unshred_edges AS (
      SELECT vertexid AS targetid,
        UNNEST(array[label_0, ..., label_l]) AS label,
        UNNEST(array[source_0, ..., source_l]) AS tmp
      FROM IncomingAdjacency
      WHERE vertexid = <?>
),
gather_edges AS (
      SELECT targetid, label, array_elements(tmp) AS sourceid
      FROM unshred_edges
      WHERE label = <?>
)
SELECT targetid, label, sourceid
FROM gather_edges
```

Listing 5.7: Example incoming neighbourhood query for a single node using the outgoing adjacency table.

**Remark 5.2**
It should be noted that up to **PostgreSQL 11**, the query optimizer was not able to optimize queries across different temporary views (CTE) that are building upon each other. This leads to the effect that the order in which the temporary views are defined determines the order in which edges are joined to the intermediate results. On the one hand this gives us leverage to control the join order, on the other hand it reduces the impact a highly sophisticated query optimizer has. **PostgreSQL 12** now has the capability to perform those optimizations. By using MATERIALIZED temporary views, we can obtain the original behavior. Even though the evaluation presented in Chapter 6 is performed on **PostgreSQL 13**, it is based on our own performance optimization using the order of CTEs. The query optimizer of **PostgreSQL 13** was **not able to handle** the complexity of the queries that are required for our evaluations and use case. □

We will use the previously described query patterns extensively in order to construct complex *SQL* queries from *Cypher* queries in the next sections.

## 5.2. Query Language Support

As previously presented in Section 5.1, *SQL* queries that have to trace along several edges soon become very lengthy and complicated, and therefore hard to maintain during the development and maintenance of an application. Consider Listing 5.8 ***just to get an impression*** of the complexity of queries that a user would have to write to use **RATG**. The example shows an implementation of *Complex Query 2* of the LDBC-SNB: "Given a start Person, find (the most recent) Messages from all of that Person's friends. Only consider Messages created before the given *maxDate* (excluding that day)" [Cou20]. In this section, we will describe an approach to use the query language *Cypher* for **RATG**.

The language *Cypher* was specifically designed to query property graph data, and therefore is much more compact and intuitively understandable. The same query as shown before in Listing 5.8 can be expressed with *Cypher* using the query depicted in Listing 5.9. In this query, the complete graph matching pattern is described in the first two lines, while the rest of the query only defines which values to return in what order.

```
MATCH (:Person {id:$personId})-[:KNOWS]-(friend:Person)
          <-[:HAS_CREATOR]-(message:Message)
WHERE message.creationDate <= $maxDate
RETURN
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  message.id AS messageId,
  CASE exists(message.content)
    WHEN true THEN message.content
    ELSE message.imageFile
  END AS messageContent,
  message.creationDate AS messageCreationDate
ORDER BY messageCreationDate DESC, toInteger(messageId) ASC
LIMIT 20
```

Listing 5.9: LDBC SNB Query 2 using Cypher.

To provide an access method with high usability, we set the goal to integrate the query language *Cypher* and make it available for **RATG**. We decided on *Cypher* over other query languages (e.g. *Gremlin* [Rod15]) due to its intuitive declarative nature, its growing support in the database community, and the fact that an open-source version is available [Gre+18].

```
WITH pattern_view_0 AS (
  WITH edge_2 AS (
    (
      WITH query1 AS (
        SELECT outgoingprimaryadjacency.vertexid AS id_0,
               UNNEST(array[...]) AS label_2, UNNEST(array[...]) AS nodeIdTmp
        FROM outgoingprimaryadjacency, vertexattributes vertexattributes0
        WHERE CAST (vertexattributes->>'id' AS bigint) = ?
          AND vertexattributes->>'Type' = 'person'
          AND vertexattributes0.vertexid=outgoingprimaryadjacency.vertexid
      ),
      query2 AS (
        SELECT query1.id_0, jsonb_array_elements_text(nodeIdTmp)::BIGINT AS id_1
        FROM query1
        WHERE label_2='KNOWS'
      )
      SELECT query2.id_0, query2.id_1 AS id_1,
             vertexattributes1.vertexattributes AS attr_1
      FROM query2, vertexattributes vertexattributes1
      WHERE (vertexattributes1.vertexattributes->>'Type' = 'person')
        AND ((vertexattributes1.vertexid=query2.id_1)
    )
      UNION
    (
      WITH query1 AS (
        SELECT incomingprimaryadjacency.vertexid AS id_0,
               UNNEST(array[...]) AS label_2, UNNEST(array[..]) AS nodeIdTmp
        FROM incomingprimaryadjacency, vertexattributes vertexattributes0
        WHERE CAST (vertexattributes->>'id' AS bigint) = ?
          AND vertexattributes->>'Type' = 'person'
          AND vertexattributes0.vertexid=incomingprimaryadjacency.vertexid
      ),
      query2 AS (
        SELECT query1.id_0, jsonb_array_elements_text(nodeIdTmp)::BIGINT AS id_1
        FROM query1
        WHERE label_2='KNOWS'
      )
      SELECT query2.id_0, query2.id_1 AS id_1,
             vertexattributes1.vertexattributes AS attr_1
      FROM query2, vertexattributes vertexattributes1
      WHERE (vertexattributes1.vertexattributes->>'Type' = 'person')
        AND ((vertexattributes1.vertexid=query2.id_1))
    )
  ),
  edge_4 AS (
    WITH query1 AS (
      SELECT incomingprimaryadjacency.vertexid AS id_1, edge_21.attr_1,
             UNNEST(array[...]) AS label_4, UNNEST(array[...]) AS nodeIdTmp
      FROM incomingprimaryadjacency, edge_2 edge_21
      WHERE edge_21.id_1=incomingprimaryadjacency.vertexid
    ),
    query2 AS (
      SELECT query1.id_1, query1.attr_1,
             jsonb_array_elements_text(nodeIdTmp)::BIGINT AS id_3
      FROM query1
      WHERE label_4='HASCREATOR'
    )
    SELECT query2.id_1, query2.attr_1, query2.id_3 AS id_3,
           vertexattributes3.vertexattributes AS attr_3
    FROM query2, vertexattributes vertexattributes3
    WHERE ((vertexattributes3.vertexattributes->>'Type' = 'post')
      OR (vertexattributes3.vertexattributes->>'Type' = 'comment'))
      AND (vertexattributes3.vertexid=query2.id_3)
      AND ((CAST (vertexattributes3.vertexattributes->>'creationDate' AS BIGINT)<=?))
  )
  SELECT DISTINCT edge_4.attr_3, edge_4.id_1, edge_4.attr_1, edge_2.id_0, edge_4.id_3
  FROM edge_4, edge_2
  WHERE (edge_4.id_1=edge_2.id_1)
)
SELECT CAST (pattern_view_0.attr_1->>'id' AS BIGINT) AS personId,
  pattern_view_0.attr_1->>'firstName' AS personFirstName,
  pattern_view_0.attr_1->>'lastName' AS personLastName,
  CAST (pattern_view_0.attr_3->>'id' AS BIGINT) AS messageId,
  CAST (pattern_view_0.attr_3->>'creationDate' AS BIGINT) AS messageDate,
  COALESCE(pattern_view_0.attr_3->>'content', pattern_view_0.attr_3->>'imageFile') AS messageContent
FROM pattern_view_0
ORDER BY CAST (pattern_view_0.attr_3->>'creationDate' AS BIGINT) DESC,
  CAST (pattern_view_0.attr_3->>'id' AS BIGINT) ASC
LIMIT 20;
```

Listing 5.8: LDBC SNB Query 2 using SQL.

### 5.2.1. Integrating an Additional Query Language

In order to easily integrate an additional query language into any software application we used the same approach as Ehlers [Ehl15] and made it accessible through a Java Database Connectivity (JDBC) interface. This way our query language mechanism can be used from any Java Software and by many standard database clients, for example *DBeaver*[1]. A first proof of concept for the integration into the JDBC interface was created by Kornev [Kor16].

Our overall integration concept for *Cypher* into **RATG** is depicted in Figure 5.10:

**1) Cypher Query Parser:** First, the query in *Cypher* syntax is parsed. To parse the statements we use the open-source parser provided by the *openCypher* project[2], which is described in [MSV17]. The parser returns an abstract syntax tree which we in turn convert into an internal graph pattern representation. The overall result of this step is an internal representation of the *Cypher* query that can be used for further processing.

**2) Internal Query Optimizer:** The internal representation of the query is then used to analyze the query structure and determine accessed attributes and returned vertices and edges. We analyze the query structure mainly in regard to the required graph pattern.

Since *Cypher* offers many ways to describe the same graph pattern, we normalize the description internally and build a single graph pattern that describes the query. Using information about the paths and their length, the optimizer decides which target tables should be used for the translation. This is necessary, since we can decide to use the edge list, the incoming, or the outgoing adjacency list for every addition of a new edge to the intermediate results. Well-known concepts like early projection can and should also be applied in this step. The result of this step is the internal representation of the query, including (among others) the order in which edges should be joined, what attributes need to be carried to the final result, and hints on what tables to use for the final translation to *SQL*.

**3) SQL Converter:** As the next step, the query is converted into a *SQL* representation of the query compatible with the target relational database system. We created prototypical implementations of the converter for **PostgreSQL** [Kor17] and optimized the translation mechanism in [Goj21]. We also created a proof of concept for **OracleDB** [Sha21].

**4) Third-party JDBC Driver:** Finally, the generated query string is handed to a third-party JDBC driver, for example the **PostgreSQL** JDBC driver. From this point on, the process is the same as with any JDBC driver. Therefore, we can use this driver in any application that expects a JDBC interface, for example we can use the graphical user interface of *DBeaver*.

In the following sections, we will describe our approach to translate *Cypher* queries into *SQL* queries.

[1] `https://dbeaver.io/`
[2] `https://www.opencypher.org/`

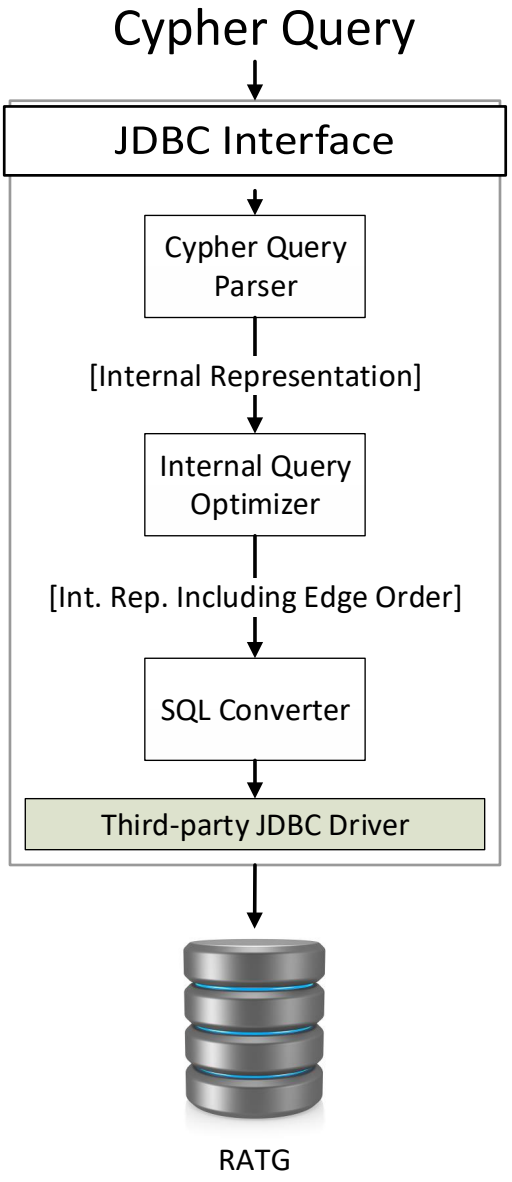



Figure 5.10.: Integration concept for the Cypher query language

To achieve the translation from *Cypher* to *SQL* statements, we need to map the available *Cypher* constructs to corresponding *SQL* features. Therefore, we first describe the general idea of *Cypher*. For this work we only consider queries that do not alter the data, hence the data is only access in a reading manner. Since *Cypher* is a very extensive query language, we will focus on the essential operations and features of the query language in this work. For detailed semantics of the *Cypher* query language see [Fra+18b] and for a detailed description of the query language itself see *The Neo4j Cypher Manual v4.0*[3].

We make the following assumptions:

1. We will present an approach to inductively construct *SQL* queries from given *Cypher* queries. Our approach makes extensive use of CTEs (which are often called temporary views) to construct the queries from preconstructed query skeletons. Therefore, we assume that we are able to use CTEs in the target database system.
2. In order to backtrack some result paths and to make sure that each edge is only used a single time for every solution, we need to have some type of array or list datatype available in the target system. This should naturally be the case, since our database schema already relies on the use of arrays to store adjacency lists.
3. Finally, to be able to translate queries that retrieve paths of variable length, we assume that the target database system offers recursive queries.

To describe our translation mechanism, we first deconstruct *Cypher* queries down to their building blocks. Then we take a look at the semantics of those elemental parts. Afterwards, we give translation skeletons to translate the *Cypher* query parts into *SQL* temporary views. We use those temporary views to then construct the whole query in *SQL*. The following sections of this chapter are:

First, we describe the most important operators of the language followed by an abbreviated description of the syntax of the language. Names of different Cypher constructs are directly taken from the *Cypher* grammar available at the **openCypher** homepage.[4]

Afterwards, we describe the semantics of parts of the *Cypher* query language: We describe the semantics of the MATCH clauses of *Cypher* in detail, while we omit to present the semantics for most of the other language constructs. We present the MATCH semantics in more detail, since this is the central concept of the query language and is the main difference to *SQL*. The other language constructs behave very similarly, if not exactly the same, as in *SQL* (ignoring syntactical differences). For the complete description of the *Cypher* semantics please consider [Fra+18b].

Finally, we present our translation of the *Cypher* MATCH clause to *SQL* and show that the translated queries return the semantically correct results. Using this central building block, we then describe how to construct entire *Cypher* queries.

[3] `https://neo4j.com/docs/cypher-manual/4.0/`
[4] `https://www.opencypher.org/`

#### 5.2.1.1. Introducing the *Cypher* Query Language

*Cypher* is designed to offer an easy and intuitive way to query data from a property graph (as defined in Definition 3.1). A *Cypher* query engine takes a *Cypher* query or clause, a property graph, and a table as input and returns a **table** containing the results of the query for the given input graph. The input table can be understood as a table containing the intermediate results. In the case of the initial *Cypher* query, the input table is a table containing a single empty tuple. The general idea is depicted in Figure 5.11.

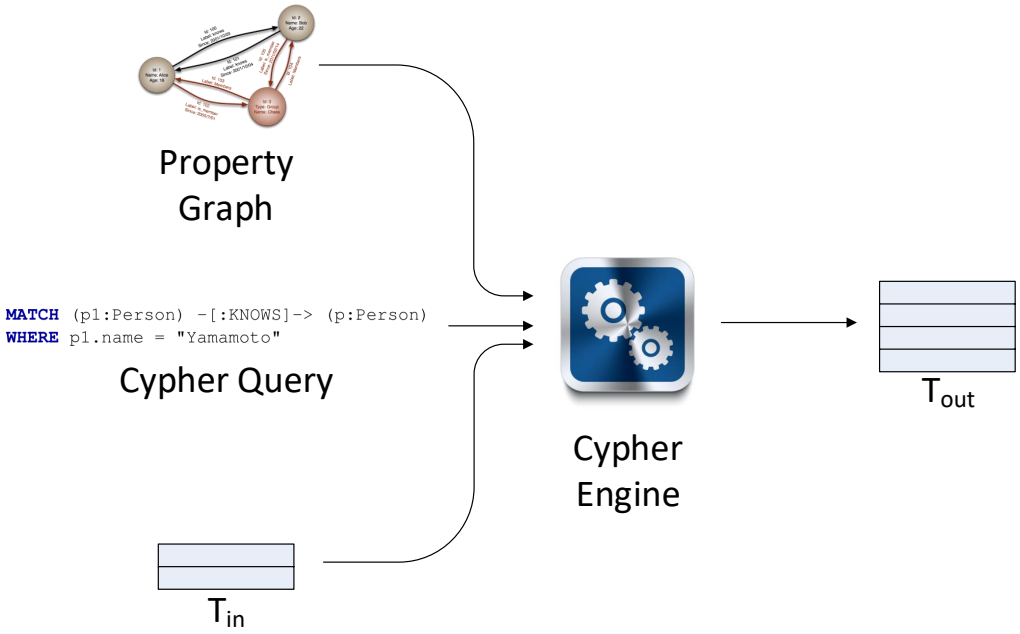



Figure 5.11.: Concept of the *Cypher* query language.

More formally, we can describe the semantics of *Cypher* by defining one relation and two functions (see [Fra+18b]):

- The *pattern matching relation* is the central piece to understand *Cypher*. This relation checks if a path $p$ in a graph $G$ satisfies a *path pattern* $\pi$ using an assignment $u$ of values to the free variables $free(\pi)$ of the pattern. We then write $(p,G,u) \models \pi$. The formal definition of the *pattern matching relation* can be found in Definition 5.43.
- The *semantics of expressions* associates an expression *expr*, a graph $G$, and a variable assignment $u$ with a value $[\![expr]\!]_{G,u}$.
- The *semantics of clauses* associates a clause $C$ and a graph $G$ with a function $[\![C]\!]_G$ that takes a table $T_{in}$ as input and returns a possibly modified table $T_{out}$. The new table can have a different number of rows or columns. Tables are defined very similary to tables in *SQL*. The exact definition can be found in [Fra+18b].

The output of a query (Figure 5.11) can then be described as the composition of those functions defining the *semantics of clauses* contained in the query. We can start the evaluation of a query by using the table that only contains one empty tuple $T_{unit}$ as input and successively applying the functions that provide the semantics for the clauses contained in the query.

In the following sections we will first describe the essential *Cypher* operators that are necessary to understand queries relevant to this work. Next, we provide a short description of the syntax, since our translation mechanism will construct the corresponding *SQL* query using the construction rules provided by the *Cypher* syntax. Afterwards we take a closer look at the semantics of *Cypher*. We will define everything necessary to understand the semantics of the *pattern matching relation*. Using the *pattern matching relation* as our central building block, we can then construct the corresponding *SQL* query that represents the MATCH clause. We will use the semantics of this relation to argue the correctness of our translation. Using our knowledge about *Cypher* syntax, we then construct the complete *SQL* query.

#### 5.2.1.2. The *Cypher* Operators

The following section shortly describes the most important operators of the *Cypher* query language. These operators are necessary to describe the graph matching patterns of *Cypher* queries, and therefore the most vital feature of the query language.

**MATCH** starts most queries and is the most essential operator of *Cypher*. Using this operator, we can define the sub-graph to search for. Those sub-graphs are defined "using ASCII-Art syntax"[5]. In Listing 5.12 a query that will match vertices that have the specified attribute value is depicted.

```
MATCH (p1:Person{name:"Yamamoto"})
```

Listing 5.12: Vertex match example.

Listing 5.13 depicts a MATCH clause that queries for all nodes of the type Person that is connected to another Person by a directed edge of type KNOWS.

```
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
```

Listing 5.13: Directed edge example.

Finally, we have to mention that it is also possible to query for paths of variable length. Listing 5.14 depicts a query that matches, starting from a specific Person, all Persons that can be reached by the use of one to three edges of type KNOWS (disregarding the direction of the edge).

```
MATCH (p1:Person{id:""})-[KNOWS*1..3]-(p2:Person)
```

Listing 5.14: Example path with variable length.

This operator can also be used in combination with the examples above, for paths including different edge types, and other cases. For more detailed view of the operator we recommend consulting Francis et al. [Fra+18b].

**WITH** is used to combine different sub-queries of a *Cypher* query by propagating selected elements to the next sub-query. The results of the WITH sub-query can then be used as a starting point for further search. Additionally, like in *SQL* the results can be aggregated, filtered, or limited before returning them. For the sake of brevity we will not discuss the required operators further and refer the interested reader to the official *Cypher* documentation. The WITH sub-query therefore fulfills the same role as a temporary view in *SQL* (or CTE in **PostgreSQL**).

```
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WITH p1, COUNT(p) AS numberOfFriends
```

Listing 5.15: WITH Example including aggregation.

[5] See https://neo4j.com/developer/cypher/

Listing 5.15 depicts a WITH sub-query that passes all Persons *p1* that *KNOW* another Person *p* with the aggregated (COUNT) number of friends (counting *p*) on to the next part of the query.

**WHERE** adopts the roles of two operators of *SQL*. Therefore, we have to distinguish the following two cases:

1. Using WHERE without a WITH clause. In this case WHERE fulfills the same role as the WHERE statement in *SQL* and filters the result applying the given logical expression before returning the result.

```
MATCH (p1:Person)
WHERE p1.name = "Yamamoto"
```

Listing 5.16: Standard WHERE usage example.

Listing 5.16 depicts an alternative to the query in Listing 5.12. Here the MATCH clause binds all vertices of type Person while the WHERE filters the matched vertices by accessing the attribute *name* and comparing it to the value *"Yamamoto"*.

2. Using WHERE in combination with/after a WITH statement. In this case, the WHERE statement acts like the HAVING statement of *SQL* and can be used to further filter the results returned by the WITH statement, including logical expressions like filtering aggregation results computed by the WITH sub-query.

```
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WITH p1, COUNT(p) AS numberOfFriends
WHERE numberOfFriends > 1
```

Listing 5.17: WHERE example functioning as HAVING.

Listing 5.17 depicts an example of WHERE in combination with the WITH clause. Here it acts like the HAVING keyword in *SQL* filtering the results based on the aggregation and only passing on *p1* that have more than one friend.

**RETURN** finishes the description of a *Cypher* query and defines the schema of the result set, which is a so-called *graph relation* [MSV17]. It is a combination of vertices, edges, and (possibly aggregated) values organized in rows. Like the results of a WITH clause, the results of a RETURN can be ordered (ORDER BY), de-duplicated (DISTINCT), and modified very similarly to *SQL*. This is done using designated keywords, which we will not discuss further for the sake of brevity.

```
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WITH p1, COUNT(p) AS numberOfFriends
WHERE numberOfFriends > 1
RETURN p1
```

Listing 5.18: Return example.

Listing 5.18 depicts a complete example query. Since the WITH clause projects the intermediate results to vertice *p1* and the result of the aggregation, the final RETURN can only access those two intermediate results columns. These are filtered further and the query only returns the vertex *p1*.

Since we now know the operators available in *Cypher* we can use these to define the *Cypher* syntax.

#### 5.2.1.3. *Cypher* Syntax

In this section we introduce part of the *Cypher* query syntax. We only introduce those parts of the grammar that are necessary for understanding this work. Later, we will use the *Cypher* syntax to construct the *SQL* query that returns the appropriate results. We will do this by using the building blocks and construction rules defined by the *Cypher* syntax.

**Cypher Query** A query consists of one or more **Single Queries**, combined with a UNION operator. The result of this query is the UNION of the results of all included **Single Queries**. The results must meet the same requirements as for the UNION operator in *SQL*. Therefore, the resulting tuples must have the same arity, the same data type, and the same name for each position. Figure 5.19 depicts a syntax diagram of the **Cypher Query**.

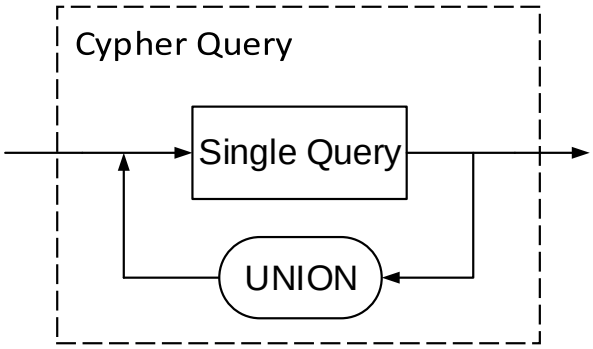



Figure 5.19.: Cypher Query Syntax.

Listing 5.20 depicts an example of a **Cypher Query** that combines two **Single Queries** using the UNION operator. Since the first **Single Query** returns vertices of the type

```
MATCH (p1:Person{name:"Yamamoto"})
WITH p1
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WITH p1, COUNT(p) AS numberOfFriends
WHERE numberOfFriends > 1
RETURN p1
        UNION
MATCH (p2:Person{name:"Silva"})
RETURN p2
```

Listing 5.20: Example cypher query including UNION.

Person the result set of the second **Single Query** must conform to this schema, and therefore can also only return vertices that follow the schema of Persons. Note that *Cypher* does not offer an intersection operator. If desired, the intersection must be achieved using WHERE clauses or other constructs.

**Single Query** A **Single Query** (see Figure 5.21) can be a **Single Part Query** or a **Multi Part Query**. Note that since each **Multi Part Query** is concluded with a **Single Part Query**, each **Single Query** query includes a **Single Part Query** as its last element (see Figure 5.24).

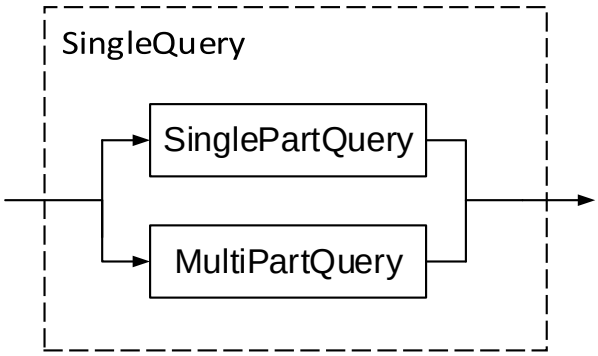



Figure 5.21.: Cypher SingleQuery Syntax.

**Single Part Query** For our purpose, a **Single Part Query** is composed of one **Reading Clause** (since we do not consider updates or function calls). The query part is finished with a RETURN statement defining the result set of the **Single Part Query**. Figure 5.23 depicts a syntax diagram of the **Single Part Query**.

Building upon Listing 5.27, we depict a **Single Part Query** that only uses the **Reading Clause** from above in Listing 5.22. The return describes that all nodes matching the requirements described in the **Reading Clause** are delivered.

**Multi Part Query** As the name suggests, a **Multi Part Query** (see Figure 5.24) consists of several parts. Since we do not consider updates, it therefore consists of one or more **Reading Clauses** each followed by a WITH clause and is concluded with a **Single**

```
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WITH p1, COUNT(p) AS numberOfFriends
WHERE numberOfFriends > 1
RETURN p1
```

Listing 5.22: Example Single Part Query.

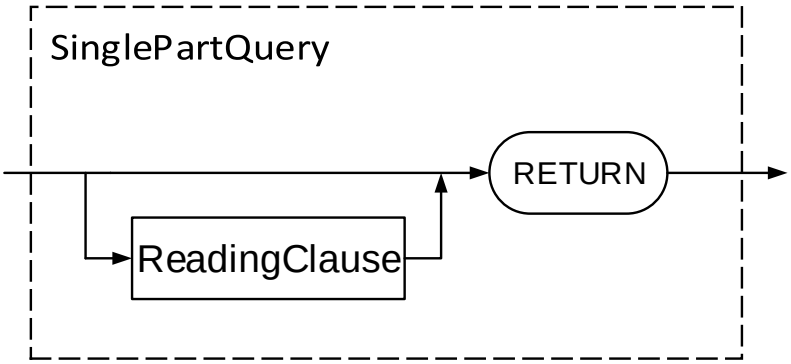



Figure 5.23.: Cypher Single Part Query Syntax.

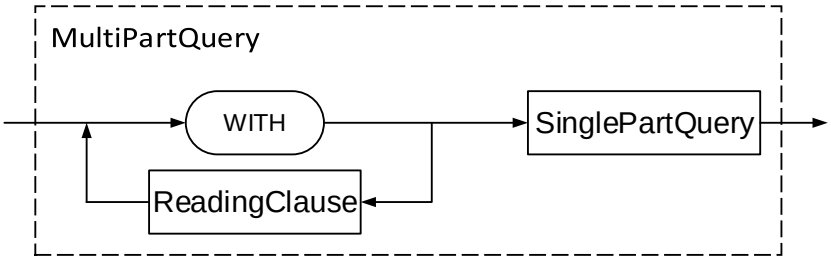



Figure 5.24.: Cypher Multi Part Query syntax.

**Part Query**. The results of the consecutive **Reading Clauses** are handed to the next clause by using WITH and can be used as input data for the next clause. For the sake of completeness, one should note that WITH can be used to deliver constants without the need of a **Reading Clause**. We will not consider this case for the rest of this work.

Listing 5.25 depicts an example where an earlier **Reading clause** that returns all vertices that have the *name "Yamamoto"* assigned is included in the **Multi Part Query**.

The second MATCH describes that the result relation of this **Reading Clause** contains the vertex *p1* and the aggregated number of vertices, which are reachable from the first vertex through KNOWS edges. In context of the WITH clause, the WHERE clause acts like the HAVING clause of *SQL* and filters the results that do not KNOW at least two vertices of type Person.

Variables are matched by name, therefore the second MATCH clause will only affect vertices that have been returned from the first MATCH clause, resulting in returning

only vertices with the *name "Yamamoto"* that *KNOW* more than one *Persons*.

```
MATCH (p1:Person{name:"Yamamoto"})
WITH p1
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WITH p1, COUNT(p) AS numberOfFriends
WHERE numberOfFriends > 1
RETURN p1
```

Listing 5.25: Example Single Query.

**Reading Clause** The **Reading Clause** is the main building block of *Cypher* queries for this work, since we do not consider update queries. These in turn use MATCH clauses as their main component. Although it is technically possible to create a *Cypher* query without a MATCH clause, it can only return constants and therefore we will not consider these types of statements in the rest of this section. Figure 5.26 depicts a syntax diagram of the **Reading Clause**.

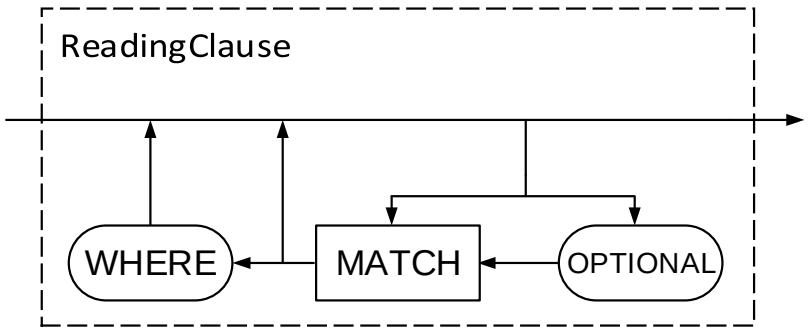



Figure 5.26.: ReadingClause syntax.

Listing 5.27 depicts an example of a simple **Reading Clause** with a single MATCH clause. The MATCH clause describes the graph patterns in which all vertices of the type Person that are connected to another vertex of type Person by using an edge of type KNOWS. The WHERE clause selects only those paths, in which the vertex $p1$ has the *name "Yamamoto"*.

```
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WHERE p1.name = "Yamamoto"
```

Listing 5.27: Example Reading Clause

**MATCH** Finally, we want to define the syntax of the **MATCH** operator. Figure 5.28 depicts the syntax of the **MATCH** operator, which consists of at least a single **node pattern** and continues with alternating **relation patterns** and **node patterns** and always ends with a **node pattern**.

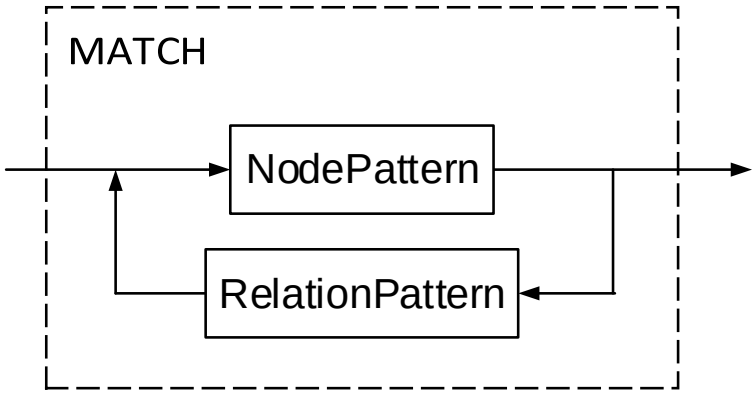



Figure 5.28.: MATCH syntax.

Now, after we have introduced the most important *Cypher* operators and syntax rules, we can describe the semantics of the *Cypher* query language:

We will describe the semantics of the MATCH clauses of *Cypher* in detail, while we omit to present the semantics for most of the other language constructs. We present the semantics of the MATCH operator in more detail, since this is the central concept of the query language and is the main difference to *SQL*. Other language constructs behave very similarly, if not exactly the same as in *SQL*. For the complete description of the *Cypher* semantics please consider [Fra+18b].

#### 5.2.1.4. *Cypher* Semantics

We now present the *Cypher* semantics defined by Francis et al. [Fra+18b] adapted to our property graph definition in Definition 3.1.

In this work we focus on the semantics of graph pattern matching, while we omit the details of handling the WHERE clause, SELECT clause, data types, lists, etc. for the purpose of brevity. These can be applied straightforward. Therefore, we first define pattern matching and semantics according to the definition in [Fra+18b]. In Section 5.2.2, we will use this definition to show the correctness of our translation mechanism.

| Concept | Instance | Set |
|---|---|---|
| Vertices/Nodes | $n$ | $N$ |
| Edges/Relationships | $r$ | $E$ |
| Label of edge e | $l$ | $L$ |
| Property keys | $k$ | $P$ |
| Names | $a$ | $\mathscr{N}$ |
| Path in the graph | $p$ | |
| Direction | $d$ | |
| Node pattern | $\chi$ | |
| Relationship pattern | $\rho$ | |
| Path pattern | $\pi$ | |

Table 5.29.: Overview of the names we use for instances and sets of concepts.

In order to define the syntax, as well as the semantics of *Cypher*, we use a number of names for different concepts. An overview of the names for concepts like vertices or edges is depicted in Table 5.29. The table contains both, the names previously defined in Definition 3.1 and new names used in this section.

| Relation/Function | Name | Domain | Codomain |
|---|---|---|---|
| Source vertex of an edge | *src* | $E$ | $N$ |
| Target vertex of an edge | *target* | $E$ | $N$ |
| Label of an edge | $\lambda$ | $E$ | $L$ |
| Properties of vertices and edges | $\sigma$ | $(N \cup E) \times P$ | $\mathscr{P}(V)$ |

Table 5.30.: Overview of the relations we use.

For look up purposes, we also give an overview of the relations and (partial) functions in Table 5.30, with the power set written as $\mathscr{P}$. We have previously defined these in Definition 3.1 and are used in this section.

We will now first define the formal syntax of the different types of patterns. Then we will define the satisfaction of patterns, which in turn is necessary to define the *pattern matching*

*function*. Later on, we can use the definition of the *pattern matching function* to define the *SQL* skeleton that we use to construct *SQL* queries representing the MATCH clause of *Cypher*.

**Formal Syntax of Patterns** We now define the syntax of patterns (see the description of MATCH in Section 5.2.1.2 and Section 5.2.1.3) in a more formal manner, which we adopted from [Fra+18b]. This is the most basic building block for our translation algorithm. The most complex pattern is a *path pattern*, which in turn consists of *node patterns* and *relationship patterns*. Therefore, we will first describe *node and relationship patterns* and then base the definition of a *path pattern* on those. After we have defined *node patterns*, *relationship patterns* and *path patterns* we will present an example including all of the above.

**Definition 5.31** [Node Pattern]
A *node pattern* $\chi$ is a triple $\chi = (a, l, K)$ where:

- $a \in \mathcal{N} \cup \{null\}$ is an optional name, where $\mathcal{N}$ is a finite set of names;
- $l \in L \cup \{null\}$ is an optional label;
- $K$ is a possibly empty finite partial mapping from the set of property keys $P$ to expressions. It is the *Cypher* equivalent for patterns to our definition of the assignment of properties to vertices and edges $\sigma$ as defined in Definition 3.1.

□

**Definition 5.32** [Free Variables of a Node Pattern]
The set of **free variables** $free(\chi)$ of the *node pattern* $\chi$ is defined as $free(\chi) = \{a\}$, if $a \neq null$, else $free(\chi) = \emptyset$. □

**Remark 5.3**
Since the following examples will require a substantial amount of indices to describe the different instances, we will use the *node pattern* (and later on *relationship pattern*) as the index to increase readability. For example, we will write $a_{\chi_1}$ to describe the aforementioned optional name $a$ that belongs to the *node pattern* $\chi_1$. □

**Example 5.33**
Let us consider the part of a *Cypher* query depicted in Listing 5.34: This MATCH clause describes the node $p1$ of the type Person, which has an attribute *name* with the value "Yamamoto".

```
MATCH (p1:Person{name:"Yamamoto"})
```

Listing 5.34: *Node pattern* example.

We can then formally describe this pattern using Definition 5.31 as the *node pattern* $\chi_1 = (a_{\chi_1}, l_{\chi_1}, K_{\chi_1})$:

- $a_{\chi_1} = p1$, identifying the matched node with the name $p1$,
- $l_{\chi_1} = Person$, defining the desired type of the node as *Person*, and
- $K_{\chi_1} = \{name \rightarrow "Yamamoto"\}$, describing that the desired value for the property key *name* is "Yamamoto".

The set of **free variables** for this *node pattern* is $free(\chi_1) = p1$. □

As we described in Section 5.2.1.3, the MATCH operator can be applied to a combination of **node patterns** and **relation patterns**. Therefore, we now define relationship patterns [Fra+18b].

**Definition 5.35** [Relationship Pattern]
A *relationship pattern* $\rho$ is a tuple $\rho = (d, a, T, K, I)$ where

- $d \in \{\rightarrow, \leftarrow, \leftrightarrow\}$ specifies the direction of the pattern (left-to-right, right-to-left, or undirected);
- $a \in \mathcal{N} \cup \{null\}$ is an optional name;
- $T \subset L$ is a possibly empty finite set of labels describing the relationship types that should be applied;
- $K$ is a possibly empty finite partial mapping from the set of property names $\mathcal{N}$ to expressions;
- $I$ is either *null* or $(m, n)$ with $m, n \in \mathbb{N} \cup \{null\}$ and $m \leq n$ describing the range of the relationship pattern (the number of hops). For $m = null$ we default to $m = 1$ and for $n = null$ we default to $n = \infty$. If $(m, n) = (null, null)$ we default to $(m, n) = (1, 1)$.

We call a relationship pattern *rigid* if its range satisfies $m = n$. □

**Definition 5.36** [Free variables of a Relationship Pattern]
The set of **free variables** $free(\rho)$ of the *relationship pattern* $\rho$ is defined as $free(\rho) = \{a\}$, if $a \neq null$, else $free(\rho) = \emptyset$. □

**Example 5.37**
The relationship pattern described in the relation part of the *Cypher* query depicted in Listing 5.38 describes a relationship $k$ of the type knows. The edge has the property *since* with the value "14.06.2018" and the direction left-to-right.

```
MATCH (p1:Person{name:"Yamamoto"})
    -[k:knows {since:"14.06.2018"}]-> () -[:likes]- (p3)
```

Listing 5.38: Relationship pattern example.

Formally, we can describe this *relationship pattern* using Definition 5.35 as
$\rho_1 = (d_{\rho_1}, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, I_{\rho_1})$ with

- $d_{\rho_1} = \rightarrow$, defining the direction left-to-right;
- $a_{\rho_1} = k$, assigning the name $k$ to the *relationship pattern*;
- $T_{\rho_1} = \{knows\}$, setting the type of the relationship to *knows*;
- $K_{\rho_1} = \{since \rightarrow \text{"14.06.2018"}\}$, making sure that only relationships are considered for which $\sigma(k, since) = \text{"14.06.2018"}$ holds;
- and $I_{\rho_1} = (null, null)$ and therefore $(m,n) = (1,1)$. This also makes this relationship pattern rigid.

The set of **free variables** for this *relationship pattern* is $free(\rho_1) = k$. □

After having defined *node patterns* and *relationship patterns*, we can now define *path patterns* [Fra+18b]:

**Definition 5.39** [Path Pattern]
A *path pattern* $\pi$ is an alternating sequence of the form

$$\chi_1 \rho_1 \chi_2 \cdots \rho_{n-1} \chi_n$$

where each $\chi_i$ is a *node pattern* and each $\rho_j$ is a *relationship pattern*. We call a *path pattern* rigid if all its *relationship patterns* are *rigid* and of *variable length* otherwise. □

**Definition 5.40** [Free Variables of a Path Pattern]
The set of free variables $free(\pi)$ of the *path pattern* $\pi$ is defined as the union of all free variables occurring in the *path pattern* $\pi$:

$$free(\pi) = \bigcup_{i \in \{1,\ldots,n\}} free(\chi_i) \cup \bigcup_{j \in \{1,\ldots,n-1\}} free(\rho_j)$$

□

After we have now defined everything we need to describe a *path pattern*, we want to take a look at the example *path pattern* depicted in Listing 5.42.

**Example 5.41**
The *path pattern* describes a path that starts at the node $p1$ of the type Person that has an attribute *name* with the value "Yamamoto". With $p1$ as its source, a relationship of the type knows, the attribute *since* with value "14.06.2018", and targets a node $p2$. This node itself is connected to a third node $p3$ by a relationship of unspecified direction and type likes.

```
MATCH (p1:Person{name:"Yamamoto"})
    -[k:knows {since:"14.06.2018"}]-> () -[:likes]- (p3)
```

Listing 5.42: Path pattern example.

Using Definitions 5.31, 5.35 and 5.39 we can formally describe this pattern as follows. Listing 5.42 depicts the *path pattern* $\chi_1\rho_1\chi_2\rho_2\chi_3$ with:

- $\chi_1 = (a_{\chi_1}, l_{\chi_1}, K_{\chi_1})$:
  - $a_{\chi_1} = p1$, identifying the matched node with the name $p1$;
  - $l_{\chi_1} = Person$, defining the desired type of the node as *Person*;
  - and $K_{\chi_1} = \{name \rightarrow "Yamamoto"\}$, describing that the desired value for the property key *name* is "Yamamoto".
- $\rho_1 = (d_{\rho_1}, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, I_{\rho_1})$ with
  - $d_{\rho_1} = \rightarrow$, defining the direction left-to-right;
  - $a_{\rho_1} = k$, assigning the name $k$ to the *relationship pattern*;
  - $T_{\rho_1} = \{knows\}$, setting the type of the relationship to *knows*;
  - $K_{\rho_1} = \{since \rightarrow "14.06.2018"\}$, making sure that only relationships are considered for which $\sigma(k, since) = 14.06.2018$ holds;
  - and $I_{\rho_1} = (null, null)$ and therefore $(m,n) = (1,1)$. This also makes this relationship pattern rigid.
- $\chi_2 = (a_{\chi_2}, l_{\chi_2}, K_{\chi_2})$ with:
  - $a_{\chi_2} = null$, not assigning any name to the described node;
  - $l_{\chi_2} = null$, not defining a type for this node;
  - and $K_{\chi_2} = \{\}$, also not making any assumptions about the properties of this node.
- $\rho_2 = (d_{\rho_2}, a_{\rho_2}, T_{\rho_2}, K_{\rho_2}, I_{\rho_2})$ with
  - $d_{\rho_2} = \leftrightarrow$, defining the direction as undirected;
  - $a_{\rho_2} = null$, assigning no name to the *relationship pattern*;
  - $T_{\rho_2} = \{likes\}$, setting the type of the relationship to *likes*;
  - $K_{\rho_2} = \{\}$, not making any assumptions about the properties of this relationship;
  - and $I_{\rho_2} = (null, null)$ and therefore $(m,n) = (1,1)$. This also makes this relationship pattern rigid.
- $\chi_3 = (a_{\chi_3}, l_{\chi_3}, K_{\chi_3})$:
  - $a_{\chi_3} = p3$ assigning the name $p3$ to the described node;
  - $l_{\chi_3} = null$ not defining a type for this node;
  - and $K_{\chi_3} = \{\}$ also not making any assumptions about the properties of this node.

And we have

$$\begin{aligned} free(\pi) &= free(\chi_1) \cup free(\rho_1) \cup free(\chi_2) \cup free(\rho_2) \cup free(\chi_3) \\ &= \{p1, k, p3\} \end{aligned}$$

as the set of free variables. □

Now that we have defined everything, we can now define the *pattern matching relation*, which corresponds to the syntax defined for the **reading clause** in Section 5.2.1.3.

**Definition 5.43** [Pattern Matching Relation]
The **pattern matching relation** $(p, G, u) \models \pi$ checks, if a path $p$ in a property graph $G$ satisfies a path pattern $\pi$ under an assignment $u$ of values to free variables of the pattern, with $u : free(\pi) \rightarrow N \cup E$ mapping the set of free variables to the combined set of nodes and edges. □

We can now take a look at the semantics of *path patterns*. After that we will examine the pattern matching relation and how we can translate it from *Cypher* to our schema in *SQL*.

**Satisfaction of *Path Patterns*** The satisfaction relation for path patterns is defined (same as by Francis et al. [Fra+18b]) with regard to a property graph $G = (N, E, \rho, \lambda, \sigma)$ (as defined in Definition 3.1), a path $p = n_1 \cdot r_1 \cdots n_{k-1} \cdot r_{k-1} \cdot n_n$ consisting of nodes $n_1, \ldots n_k \in N$ and relationships $r_1, \ldots, r_{k-1} \in E$ and an assignment $u$ in compliance to our definition of a property graph.

**Definition 5.44** [Satisfaction of Node Patterns]
Let $p$ be a path in which all relationships are *distinct*, with $G = (N, E, \rho, \lambda, \sigma)$ be a property graph, and $u$ be an assignment. The relationships contained in a path are *distinct*, if the path does not contain any relationship more than a single time.

We define the satisfaction of rigid patterns inductively, with the base case given by node patterns. Let $\chi = (a, l, K)$ be a node pattern within the path pattern $\pi$, then the path $(n, G, u) \models \chi$ if all the following hold:

- either $a$ is *null* or $u(a) = n$,
  meaning either a is not named or $u$ assigns $n$ to $a$;
- $l = \sigma(n, type)$,
  making sure the label of the vertex in the node pattern matches the type of the node in the graph. Since we do not directly assign types to nodes in our definition of the property graph, we use the reserved property key 'type' for this purpose (see Definition 3.1);
- and $[\![\sigma(n,k) = K(k)]\!]_{G,u} = true$ for each $k$ such that $K(k)$ is defined in $\chi$,
  checking if all properties defined in the pattern match the assigned vertex.

We then say the *node n* **satisfies** the *path pattern* $\chi$ in the graph $G$ with the assignment $u$. A *node pattern* is the smallest valid *path pattern*. □

**Remark 5.4**
The property that the relationships in a path have to be distinct is a characteristic of *Cypher*. Due to this, the resulting paths obtained from a *Cypher* query can never contain the same relation twice, but can still contain the same node several times. □

Let us revisit the example *path pattern* $\chi_1\rho_1\chi_2\rho_2\chi_3$ described in Example 5.41 obtained from Listing 5.42 with $\chi_1 = (p1, Person, \{name \to Yamamoto\})$, $\chi_2 = (null, null, \{\})$, and $\chi_3 = (p3, null, \{\})$.

**Example 5.45**
By assigning the unique name $p2$ to $\chi_2$ we obtain $\chi_1 = (p1, Person, \{name \to Yamamoto\})$, $\chi_2 = (p2, null, \{\})$, and $\chi_3 = (p3, null, \{\})$.

Let $G = (N, E, \rho, \lambda, \sigma)$ be the graph defined in Example 3.2 with $N = \{n_1, n_2, n_3, n_7, n_{13}\}$ and $\sigma(n_1, type) = Person, \sigma(n_1, name) = Yamamoto, \ldots$.

Using Definition 5.44 (rigid satisfaction of node patterns) we can infer from the graph that

$$
\begin{array}{lll}
(n_1, G, u_1^{\chi_1}) \models \chi_1 & (n_1, G, u_1^{\chi_2}) \models \chi_2 & (n_1, G, u_1^{\chi_3}) \models \chi_3 \\
(n_2, G, u_2^{\chi_1}) \not\models \chi_1 & (n_2, G, u_2^{\chi_2}) \models \chi_2 & (n_2, G, u_2^{\chi_3}) \models \chi_3 \\
(n_3, G, u_3^{\chi_1}) \not\models \chi_1 & (n_3, G, u_3^{\chi_2}) \models \chi_2 & (n_3, G, u_3^{\chi_3}) \models \chi_3 \\
(n_7, G, u_7^{\chi_1}) \not\models \chi_1 & (n_7, G, u_7^{\chi_2}) \models \chi_2 & (n_7, G, u_7^{\chi_3}) \models \chi_3 \\
(n_{13}, G, u_{13}^{\chi_1}) \not\models \chi_1 & (n_{13}, G, u_{13}^{\chi_2}) \models \chi_2 & (n_{13}, G, u_{13}^{\chi_3}) \models \chi_3
\end{array}
$$

hold for mappings $u_k^{\chi_k}(p_i) = n_k$ defined as described above. As one can see, for $\chi_1$ only $n_1$ qualifies, while for $\chi_2$ and $\chi_3$ any node can be chosen. □

**Definition 5.46** [Satisfaction of Rigid Paths]
Let $p$ be a path in which all relationships are *distinct*, $G = (N, E, \rho, \lambda, \sigma)$ be a property graph, $u$ be an assignment, and list() be the list constructor.

Also let $\chi$ be a node pattern for the inductive case, let $\pi$ be a rigid path pattern, and let $\rho = (d, a, T, K, I)$ be a rigid relationship pattern. Since $\rho$ is rigid, $I$ is $(m, m)$ with $m \in \mathbb{N}_0$. We distinguish two cases:

1. For $m = 0$, we have that $(n \cdot \mathit{remainder}, G, u) \models \chi\rho\pi$, if

   a) $a$ is *null* or $a = \mathit{list}()$;

   b) and $(n, G, u) \models \chi$ and $(\mathit{remainder}, G, u) \models \pi$.

2. For $m \geq 1$, we have that $(n_1 \cdots r_m n_{m+1} \cdot \mathit{remainder}, G, u) \models \chi\rho\pi$, if **all** of the following hold for every $i \in \{1, \ldots, m\}$:

   a) either $a$ is *null* or $u(a) = \mathit{list}(r_1, \ldots, r_m)$,
   meaning that $a$ is not named or $u$ assigns exactly the first $m$ relationships of the path to $a$;

   b) $(n_1, G, u) \models \chi$ and $(\mathit{remainder}, G, u) \models \pi$,
   matching the first node with the first *node pattern* $\chi$ and the remainder of the input path matches the remaining *node and relationship patterns* of the *path pattern* $\pi$;

   c) $\lambda(r_i) \in T$,
   checking if the label of all relations fit the label of the *relationship pattern* $\rho$;

   d) $[\![\sigma(r_i, k) = K(k)]\!]_{G_u} = \mathit{true}$ for every key $k$ such that $K(k)$ is defined in the *relationship pattern* $\rho$,
   checking if all expressions defined in the *relationship pattern* fit the given relationships of the path in the graph;

e) $(src(r_i), target(r_i)) \begin{cases} (n_i, n_{i+1}), (n_{i+1}, n_i) & \text{if } d \text{ is } \leftrightarrow, \\ (n_i, n_{i+1}) & \text{if } d \text{ is } \rightarrow, \\ (n_{i+1}, n_i) & \text{if } d \text{ is } \leftarrow . \end{cases}$
last, making sure that the direction of the relationship conforms to the direction of the *relationship pattern* $\rho$.

□

**Example 5.47**
Let us now take a look at the remaining two *relationship patterns* of the *path pattern* $\chi_1\rho_1\chi_2\rho_2\chi_3$ that we have considered in Example 5.45. The same as for nodes, we assign the unique name $l$ to $p_2$ and obtain the following two *relationship patterns*:

- $\rho_1 = (d_{\rho_1}, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, I_{\rho_1})$ with
  - $d_{\rho_1} = \rightarrow$, defining the direction of this relationship pattern as left-to-right;
  - $a_{\rho_1} = k$, assigning the name $k$ to this relationship pattern;
  - $T_{\rho_1} = \{knows\}$, setting the label of the matched relations to *knows*;
  - $K_{\rho_1} = \{since \rightarrow$ "14.06.2018"$\}$ restricting matched relations to those that have the value "14.06.2018" assigned to the key "since";
  - and $I_{\rho_1} = (null, null)$ and therefore $(m,n) = (1,1)$ which defines this pattern as exactly one edge hop.
- $\rho_2 = (d_{\rho_2}, a_{\rho_2}, T_{\rho_2}, K_{\rho_2}, I_{\rho_2})$ with
  - $d_{\rho_2} = \leftrightarrow$, setting the direction of the pattern to undirected (or both directions);
  - $a_{\rho_2} = l$, defining the name of this pattern as $l$;
  - $T_{\rho_2} = \{likes\}$, restricting the matched relations to those that are of the type *likes*;
  - $K_{\rho_2} = \{\}$, not defining any restrictions on property values;
  - and $I_{\rho_2} = (null, null)$ and therefore $(m,n) = (1,1)$ to only match a path of length 1.

Let $G = (N, E, \rho, \lambda, \sigma)$ be the graph defined in Example 3.2 with

- $N = \{n_1, n_2, n_3, n_7, n_{13}\}$
- $E = \{r_4, r_5, r_6, r_{11}, r_{12}, r_{14}\}$
- $\rho = \{r_4 \rightarrow (n_1, n_2), r_5 \rightarrow (n_1, n_3), r_6 \rightarrow (n_2, n_7), r_{11} \rightarrow (n_{13}, n_1), r_{12} \rightarrow (n_1, n_7),$ $r_{14} \rightarrow (n_{13}, n_7)\}$
- $\lambda = \{r_4 \rightarrow knows, r_5 \rightarrow knows, r_6 \rightarrow likes, r_{11} \rightarrow hasCreator, r_{12} \rightarrow likes, r_{14} \rightarrow replyOf\}$
- $\ldots \sigma(r_4, since) = 14.06.2018$

We know that the only suitable choice for $\chi_1$ is $n_1$ and because of $d_{\rho_1} = \rightarrow$ the only suitable edge instances are $r_4, r_5$ and $r_{12}$. Since only $\lambda(r_4), \lambda(r_5) \in T_{\rho_1}$, we can exclude $r_{12}$ from the list of suitable edges for $\rho_1$. Finally, only $\sigma(r_4, since) = K_{\rho_1}(since) = 14.06.2018$.

Therefore, the only possible solution, assuming there exist $r_?$ and $n_?$ with $(\pi_2(target(r_4) \cdot r_? \cdot n_?, G, u) \models \pi)$ is $(n_1 \cdot r_4 \cdot n_2 \cdot r_? \cdot n_?, G, u) \models \chi_1\rho_1\pi$. This also fixates $n_2$ for $\chi_2$, since otherwise this would not represent a path in the graph.

The task remains to find all paths $(n_2 \cdot r_? \cdot n_?) \models \pi$, which is the same as $(n_2 \cdot r_? \cdot n_?) \models \chi_2\rho_2\chi_3$. First, we have to makes sure that $(n_2, G, u) \models \chi_2$, which we know to be true from Example 5.45.

Since $d_{\rho_2} = \leftrightarrow$, we can only choose from the two edges $r_4$ and $r_6$ to construct a suitable path. Any edge is allowed to occur a single time in a path. This leaves us with $r_6$ as the only option. We know that $\lambda(r_6) = knows \in T_{\rho_2} = \{likes\}$ and that there is no constraint on the attributes of the relationship. This leaves us with $n_7$ as the only option for $\chi_3$.

At last, we need to show that $(n_7, G, u) \models \chi_3$, which we have already demonstrated in Example 5.45.

Therefore, we can overall conclude that $n_1 \cdot r_4 \cdot n_2 \cdot r_6 \cdot n_7$ is the only path that satisfies the given *path pattern* $\chi_1\rho_1\chi_2\rho_2\chi_3$ given a suitable assignment $u$:

$$(n_1 \cdot r_4 \cdot n_2 \cdot r_6 \cdot n_7, G, u) \models \chi_1\rho_1\chi_2\rho_2\chi_3$$

with $G = (N, E, \rho, \lambda, \sigma)$ and $u = \{p_1 \rightarrow n_1, k \rightarrow r_4, p_2 \rightarrow n_2, l \rightarrow r_6, p_3 \rightarrow n_7\}$ □

**Satisfaction of Variable Length *Path Patterns*** Up to now, we have only considered rigid *relationship patterns*. Therefore, we now define the satisfaction of variable length *relationship patterns* based on the definition of the satisfaction of rigid patterns [Fra+18a].

**Definition 5.48** [Satisfaction of Paths of Variable Lengths]
Let $\rho = (d, a, T, K, (m, n))$ be a *relationship pattern* with $m, n \in \mathbb{N}$ and $m < n$. We then call $\rho$ a relationship pattern of **variable length**. A *path pattern* is called of **variable length**, if it includes a *relationship pattern* of variable length.

Let $\rho' = (d, a, T, K, (m', m'))$ be a rigid *relationship pattern*. We then say that $\rho$ subsumes $\rho'$ ($\rho \sqsupset \rho'$), if $m' \in [m, n]$. This subsumption is extended to variable length *path patterns*:

Let $\pi = \chi_1\rho_1\chi_2 \ldots \chi_{k-1}\rho_{k-1}\chi_k$ be a *path pattern* of variable length and
$\pi' = \chi_1\rho'_1\chi_2 \ldots \chi_{k-1}\rho'_{k-1}\chi_k$ be a rigid *path pattern*, then $\pi \sqsupset \pi'$ if $\rho_i \sqsupset \rho'_i$ for every $i \in \{1, \ldots, k-1\}$.

Then the rigid extension of $\pi$ is defined as

$$rigid(\pi) = \{\pi' | \pi' \text{ is rigid and } \pi \sqsupset \pi'\}$$

Finally, we know for any $\pi'$ and $(p,G,u)$, that if $(p,G,u) \models \pi'$ then also $(p,G,u) \models \pi$ holds. □

**Example 5.49**
Let's consider the modified example in Listing 5.50.

```
MATCH (p1:Person{name:"Yamamoto"})
    -[k:knows|likes*1..2]-> () -[:likes]- (p3)
```

Listing 5.50: Relationship pattern with variable length.

Then we have the *path pattern* $\pi = \chi_1\rho_1\chi_2\rho_2\chi_3$ with the *relationship patterns*

$$\rho_1 = (\rightarrow, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, I_{\rho_1}), \qquad \text{with } I_{\rho_1} = (1,2) \text{ and}$$
$$\rho_2 = (\leftrightarrow, a_{\rho_2}, T_{\rho_2}, K_{\rho_2}, I_{\rho_2}), \qquad \text{with } I_{\rho_2} = (1,1).$$

Using Definition 5.48 and the fact that $\rho_2$ is rigid, we can see that the only *relationship pattern* that $\rho_2$ subsumes is $\rho_2$: $\rho_2 \sqsupset \rho_2$.

Because $I_{\rho_1} = (1,2)$, the only two possible *relationship patterns* that $\rho_1$ subsumes are ${\rho_1}' = (\rightarrow, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, (1,1))$ and ${\rho_1}'' = (\rightarrow, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, (2,2))$. Therefore, we can find two rigid *path patterns* that are subsumed by $\pi$:

$$\pi' = \chi_1{\rho_1}'\chi_2\rho_2\chi_3 \qquad \text{with } \pi \sqsupset \pi'$$
$$\pi'' = \chi_1{\rho_1}''\chi_2\rho_2\chi_3 \qquad \text{with } \pi \sqsupset \pi''$$

Therefore, $rigid(\pi) = \{\pi', \pi''\}$.

□

**The Match Operator for *Path Patterns*** After having defined all the required components of a pattern, we can now define the *pattern matching* function the same way as it is defined in [Fra+18b].

**Definition 5.51** [Match Operator for Path Patterns]
For a *path pattern* $\pi$ and a graph $G$, we define

$$match(\pi, G, u) = \biguplus_{\substack{p \in G \\ \pi' \in rigid(\pi)}} \{^{dom(u')=free(\pi)-dom(u) \text{ and } u \text{ such that } (p,G,u\cdot u') \models \pi'}\}$$

We use $\biguplus$ as the bag union (same as in *SQL*). This means the result can possibly contain a solution for $u$ several times. □

We use the *path pattern* $\pi$, graph $G$, and an input variable assignment $u$ that contains the intermediate results forwarded to this *match operator*. The additional conditions make sure that the already defined variable assignments are not changed and that the new variable assignment ($u$ extended with the assignment $u$') satisfies the current *path pattern*.

Finally, we can go back to the semantics of clauses, which we have described in Section 5.2.1.1 and take a closer look at the semantics of the MATCH clause.

$$[\![\text{MATCH}(\pi)]\!]_G(T) = \biguplus_{u \in T} \{u \cdot u' | u' \in match(\pi, G, u)\}$$

The semantics of a MATCH clause take an input table $T$, a *path pattern* $\pi$, and a graph $G$ and returns a table. As mentioned before, the tuples in the table represent the input variable assignments. Each of the input variable assignments is extended with viable resulting variable assignments $u$' computed by using the definition of the *match operator for path patterns*. We thereby extend the intermediate results with new columns that represent the variable names that correspond to names occurring in the pattern, but do not yet occur in the intermediate result or with new tuples.

### 5.2.2. General Approach to Translating *Cypher* Queries

To translate *Cypher* queries we leverage the concept introduced in Section 5.2.1.1. The input for evaluating a *Cypher* clause (respectively query) is a table that contains the already existing intermediate results, a property graph, and the *Cypher* clause $C$.

We model the semantic function that represents the *semantics of clause C* with the creation of an associated CTE that uses the intermediate results $T_{in}$ (in form of another CTE) in the FROM clause, the graph in the FROM clause (in form of the graph's different tables), and encode the *Cypher* clause in form of an *SQL* query. Then the *SQL* query engine will compute the output table $T_{out}$ as the result of the CTE. In turn, the output table can then be used as the input table for the next clause.

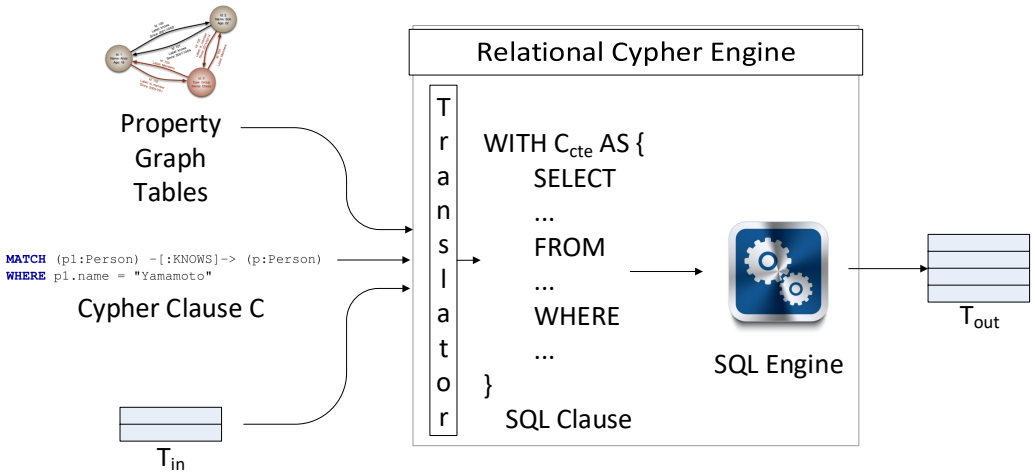



Figure 5.52.: General translation concept.

Same as previously described in Section 5.2.1.1, we then let the RDBMS compute the output of a *Cypher* query by composing the CTEs representing the clauses in order to obtain a CTE that represents the complete query. To do this, we gradually build up the complete query by constructing CTEs. Each CTE represents a clause or a combination of those using the syntax rules presented in Section 5.2.1.3 and takes the previous CTE as input. This input CTE represents the input table $T_{in}$ for the *Cypher* query engine, the database instance represents the property graph, and the *Cypher* clause is encoded in the CTE itself.

We will now describe the construction rules that we use to create the CTEs described in this section and argue their correctness.

#### 5.2.2.1. Translating *Cypher* Match Clauses

We construct a *SQL* query that returns all the paths satisfying the given *path pattern* using Definitions 5.44 and 5.46, which define when a given path satisfies a *path pattern*. We let the RDBMS compute the mapping $u$. The resulting mapping is represented by the output table $T_{out}$ in Figure 5.52.

In order to build a CTE that describes the complete set of MATCH clauses, we reduce the construction of the complete clause to the composition of CTEs that describe the *node patterns* and *relationship* patterns used in the MATCH clauses. This means we have to construct a temporary view for every *node and relationship pattern*.

We will reduce the translation of *rigid relationship patterns* to patterns of length 1, since we can rewrite any rigid *Cypher* query into a *Cypher* query that only uses *relationship patterns* of length 1.

**Example 5.53**
For example, consider the *Cypher* query in Listing 5.54: The query returns the persons's *firstName* together with the *id* of posts that friends of a friend like. One can easily see that the query depicted in Listing 5.55 will return the same results, but only uses *relationship patterns* of length 1.

```
MATCH (p:Person) -[:KNOWS*2]- (m:Person) -[:LIKES]-> (o:Post)
RETURN p.firstName, o.id
```

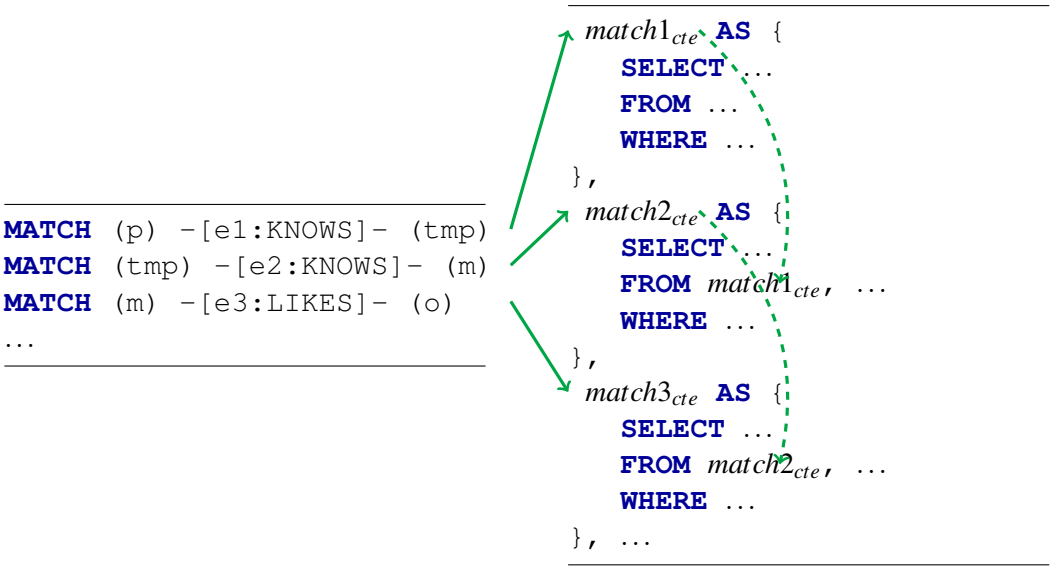
Listing 5.54: Rigid *Cypher* Query Example.

```
MATCH (p:Person) -[:KNOWS]- (tmp)
MATCH (tmp) -[:KNOWS]- (m:Person)
MATCH (m:Person) -[:LIKES]-> (o:Post)
RETURN p.firstName, o.id
```

Listing 5.55: Split MATCH Clause Example.

□

The general idea of the construction of *SQL* queries that correspond to MATCH clauses is depicted in Listing 5.56. We translate each of the MATCH clauses into a separate CTE that represents this single clause. We then use the already constructed CTE as input table for the next CTE as can be seen in Figure 5.52, since the table resulting from the previous CTE already contains bindings. These bindings will be used to compute further bindings required for the results of the query. Then the results received by evaluating the last CTE contains bindings that fulfill all requirements defined by the MATCH clauses of the query.




Listing 5.56: General Idea of the Cypher Match Clause Translation.

We will now first describe how to translate *node patterns*, since those are the smallest components required. Afterwards, using the previously described translation of *node patterns*, we will describe the translation of *rigid path patterns* and then the translation of *path patterns of variable length*.

**Translating *Node Patterns*** We first take a look at the translation of *node patterns* according to Definition 5.44: Let $\chi = (a, l, K)$ be a node pattern and $k_i$ with $i \in 1, \ldots, n$ all keys for which $K(k_i)$ is defined. We assume that $a \neq null$, otherwise we assign a unique name that is not yet in use.

We construct the corresponding CTE by using the skeleton in pseudo *SQL* code depicted in Listing 5.57.

```
WITH χ_cte AS (
  SELECT a.vid AS χvid, a.attributes AS χatt
  FROM vertices a
  WHERE a.attributes(type) = l
    AND a.attributes(k_1) = K(k_1)
    ... AND a.attributes(k_n) = K(k_n)
)
```

Listing 5.57: Node pattern translation skeleton.

For simplicity (ignoring efficiency at this point), we select everything from the vertex table. At the same time, we restrict the selection under consideration of the type of the vertex. Then we add a filter for every occurrence of a constraint $K(k)$ in $\chi$.

Each row returned by this CTE represents one $(n, G, u)$ for which $(n, G, u) \models (a, l, K)$ holds:

- $n = \chi vid$: Each returned row represents one node;
- $G$ is the database instance representing the graph;
- and $u(a) = n$: The node $n$ is bound to the given variable name $a$.

Then the results of this CTE trivially represent all nodes satisfying $\chi$.

**Example 5.58**
By applying the skeleton CTE described before, we obtain the CTEs shown in Listings 5.59 and 5.61.

```
WITH χ_1cte AS (
  SELECT p_1.vid AS χ_1vid, p_1.attributes AS χ_1att
  FROM vertices p_1
  WHERE p_1.attributes(type) = Person
    AND p_1.attributes(name) = Yamamoto
)
```

Listing 5.59: Node pattern for $\chi_1$.

As we can see, $\chi_{1_{cte}}$ (depicted in Table 5.60) will only return the tuple representing $n_1$, since this is the only vertex of the type Person that has *"Yamamoto"* assigned to the key *name*.

| $\chi_1 vid$ | $\chi_1 att$ |
|---|---|
| 1 | {"type": "Person", "name": "Yamamoto", "firstName": ...} |

Table 5.60.: The output of $\chi_{1_{cte}}$.

```
WITH χ2cte AS (
  SELECT p2.vid AS χ2vid, p2.attributes AS χ2att
  FROM vertices p2
)
```

Listing 5.61: Node pattern for $\chi_2$.

```
WITH χ3cte AS (
  SELECT p3.vid AS χ3vid, p3.attributes AS χ3att
  FROM vertices p3
)
```

Listing 5.62: Node pattern for $\chi_3$.

At the same time, $\chi_{2_{cte}}$ and $\chi_{3_{cte}}$ both return the whole vertex table (depicted in Table 5.63). Therefore, all three CTEs return exactly the nodes that satisfy the given *node patterns*.

| $\chi_2 vid$ / $\chi_3$vid | $\chi_2 att$ / $\chi_3 att$ |
|---|---|
| 1 | {"type": "Person", "name": "Yamamoto", "firstName": "Akira"} |
| 2 | {"type": "Person", "name": "Silva", "firstName": "Ana"} |
| 3 | {"type": "Person", "name": "Lepland", "firstName": "Carmen"} |
| 7 | {"type": "Post", "creationDate": "03.03.2020"} |
| 13 | {"type": "Post", "creationDate": "02.03.2020"} |

Table 5.63.: The output of $\chi_{2_{cte}}$. / $\chi_{3_{cte}}$.

□

**Translating *Rigid Patterns*** Second, we consider the translation of *rigid path patterns* that include *relationship patterns* according to Definition 5.46.

Let $\chi$ be a node pattern that we have already translated, let $\pi$ be a rigid path pattern, and let $\rho = (d, a, T, K, I)$ be a relationship pattern in the *path pattern* $\pi_{complete} = \chi\rho\pi$. Since $\rho$ is rigid, $I$ is $(m, m)$ with $m \in \mathbb{N}_0$.

Assume $m > 1$, then we can transform $\rho = (d, a, T, K, I)$ into the *path pattern* $\chi\rho_1\chi_2 \ldots \chi_m\rho_m\pi$ with $\rho_i = (d, a, T, K, (1,1))$, $i \in \{1, \ldots, m\}$, and $\chi_j = (null, null, \{\})$ with $j \in \{2, \ldots, m\}$. This means we split the *relationship pattern* of length $m$ into m separate *relationship patterns* of length 1, not further restricting the intermediate nodes. We can then inductively construct a *SQL* query that computes all paths that satisfy $\chi\rho_1\chi_2 \ldots \chi_m\rho_m\pi$. We also assume that $a_i \neq null$, otherwise we assign a unique name.

Consequently, we can assume $m = 1$ for the translation of ***rigid*** *path patterns*. For the sake of simplicity, let us assume the direction $d = \rightarrow$. The case $d = \leftarrow$ can analogously be computed using the incoming adjacency table or the edge list table (by switching source and target). Also, if the direction is $d = \leftrightarrow$, we construct the CTE for both directions and create the UNION of both results.

For the base case of the translation of *relationship patterns* we have two different options.

- Assuming the direction is $d = \rightarrow$, the straight forward method is to use the edge list table as depicted in Listing 5.64.
- Assuming the direction is $d = \rightarrow$ and $K = \emptyset$ defines that we do not have any restrictions on the properties of the edge, so we can alternatively use the outgoing adjacency table. This is depicted in Listing 5.65.

The CTEs representing the base case returns only results that satisfy the given (part of the) *path pattern*, because:

- We assume $a \neq null$. This means $u(a) = list(r_1)$ must hold (since $m = 1$). We can ensure this by setting $u(a) = \rho_1 eid$;
- $(n_1, G, u) \models \chi$ follows by construction of $\chi_{cte}$ and $(remainder, G, u) \models \pi$ follows by construction of the remaining *path pattern*;
- $\lambda(r_i) \in T$ holds, because of $label \in T$;
- $[\![\sigma(r_i, k) = K(k)]\!]_{G_u} = true$ holds, because of $attributes(k_i) = K(k_i)$;
- and $(src(r_i), target(r_i)) = (n_i, n_{i+1})$ holds, because of $sid = \chi_1 vid$ (and respectively $vertexid = \chi_1 vid$ for the adjacency list skeleton) and $tid = \chi_2 vid$ (respectively $\rho_1 target = \chi_2 vid$).

After we have translated the starting point, we iteratively construct the rest of the *path pattern*. As before, we can choose between the edge list table and the adjacency tables to translate the next *relationship pattern*.

```
WITH ρ1cte AS (
  SELECT
    sid AS ρ1source,
      χ1att,
        eid AS ρ1eid, label AS ρ1label, att AS ρ1att,
      χ2att,
    tid AS ρ1target
  FROM χ1cte, edges, χ2cte
  WHERE sid = χ1vid
    AND tid = χ2vid
    AND label ∈ T
    AND attributes(k1) = K(k1)
        ... AND attributes(kn) = K(kn)
)
```

Listing 5.64: The *relationship pattern* skeleton using the edge list table for the base case.

```
WITH unshred_edges AS (
  SELECT vertexid AS ρ1source, χ1att
        UNNEST(array[label_0, ..., label_k]) AS label,
        UNNEST(array[eid_0, ..., eid_k]) AS eidTmp,
        UNNEST(array[target_0, ..., target_k]) AS tmp
  FROM χ1cte, outgoing_adjacency
  WHERE vertexid = χ1vid
),
gather_edges AS (
  SELECT ρ1source, χ1att, array_elements(eidTmp) AS ρ1eid,
    label AS ρ1label, array_elements(tmp) AS ρ1target
  FROM unshred_edges
  WHERE label ∈ T
),
ρ1cte AS (
  SELECT
    ρ1source,
      χ1att,
        ρ1eid, ρ1label,
      χ2att,
    ρ1target
  FROM gather_edges, χ2cte
  WHERE ρ1target = χ2vid
)
```

Listing 5.65: The *relationship pattern* skeleton using the adjacency list table for the base case.

- Let us assume again that $d = \rightarrow$. As before, we can analogously translate the pattern for $d = \leftarrow$ by switching the source and target ids of the edge list table. The inductive case using the edge list is depicted in Listing 5.66.
- Assuming $d = \rightarrow$ and $K = \emptyset$, we can alternatively use the outgoing adjacency table. This is depicted in Listing 5.67. If $d = \leftarrow$, we analogously use the incoming adjacency table.

The results of the inductively constructed CTE satify the given *path pattern*, because:

- We assume $a \neq null$. This means $u(a) = list(r_1)$ must hold (since $m = 1$). We achieve this by defining $u(a) = \rho_i eid$;
- $(n_{i-1}, G, u) \models \chi_{i-1_{cte}}$ follows by construction of $\chi_{i-1_{cte}}$ and $(remainder, G, u) \models \pi$ follows by construction of the remaining *path pattern*;
- $\lambda(r_i) \in T$ holds because of $label \in T$;
- $[\![\sigma(r_i, k) = P(k)]\!]_{G_u} = true$ holds because of $attributes(k_i) = K(k_i)$;
- and $(src(r_i), target(r_i)) = (n_i, n_{i+1})$ holds because of $sid = \rho_{i-1} target$ (respectively $vertexid = \rho_{i-1} target$) and $\rho_1 target = \chi_{i+1} vid$.

The overall finished translation for the pattern $\pi_{complete}$ is depicted in Listing 5.68. Note that we need to add a final set of conditions to make sure that none of the resulting edge translations use the same edge twice, since this is restricted by Definition 5.46.

```
WITH match(π, G) AS (
  SELECT *
  FROM ρ_{k_cte}
  WHERE ∀i, j ∈ {1,...,k}, i ≠ j : (ρ_i eid ≠ ρ_j eid)
)
```

Listing 5.68: The complete match skeleton.

**Remark 5.5**
The structure of both ways to translate a *relationship pattern* is exactly the same. This means we are not restricted by the decision which type of table to use. Therefore, we can use both the edge list and adjacency lists to translate a single *path pattern*. This gives us a huge amount of chances for optimization later on.

Also, we do not consider the case $m = 0$ here, since we can simply return the translation of $\chi$ to compute the results. □

```
WITH ρ_{i_cte} AS (
  SELECT
    ρ_{i−1_cte}.*,
      eid AS ρ_i eid, label AS ρ_i label,
      attributes AS ρ_i att,
    χ_{i+1_cte}.attributes AS χ_{i+1} att,
    tid AS ρ_i target
  FROM ρ_{i−1_cte}, edges, χ_{i+1_cte}
  WHERE sid = ρ_{i−1} target
    AND tid = χ_{i+1} vid
    AND label ∈ T
    AND attributes(k_1) = K(k_1)
      ... AND attributes(k_n) = K(k_n)
)
```

Listing 5.66: The *relationship pattern* skeleton using the edge list table for the inductive case.

```
WITH unshred_edges AS (
  SELECT ρ_{i−1_cte}.*,
         UNNEST(array[label_0, ..., label_k]) AS label,
         UNNEST(array[eid_0, ..., eid_k]) AS eidTmp,
         UNNEST(array[target_0, ..., target_k]) AS tmp
  FROM ρ_{i−1_cte}, outgoing_adjacency
  WHERE vertexid = ρ_{i−1} target
),
gather_edges AS (
  SELECT ρ_{i−1_cte}.*, array_elements(eidTmp) AS ρ_i eid,
    label AS ρ_i label, array_elements(tmp) AS ρ_i target
  FROM unshred_edges
  WHERE label ∈ T
),
ρ_{i_cte} AS (
  SELECT
    ρ_{i−1_cte}.*,
        ρ_i eid, ρ_i label,
      χ_{i+1} att,
    ρ_i target
  FROM gather_edges, χ_{i+1_cte}
  WHERE ρ_i target = χ_{i+1} vid
)
```

Listing 5.67: The *relationship pattern* skeleton using the adjacency list table for the inductive case.

**Example 5.69**
The goal is to translate the **rigid** *path pattern* $\chi_1\rho_1\chi_2\rho_2\chi_3$ of which we have already translated $\chi_1, \chi_2$, and $\chi_3$ and therefore have the CTEs $\chi_{1_{cte}}, \chi_{2_{cte}}$, and $\chi_{3_{cte}}$ available.

First, we start translating the base case $\chi_1\rho_1\chi_2$ with $\rho_1 = (d_{\rho_1}, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, I_{\rho_1})$ where

- $d_{\rho_1} = \rightarrow$ is the direction left-to-right;
- $a_{\rho_1} = k$ defines the name of this pattern as $k$;
- $T_{\rho_1} = \{knows\}$ restricts the relations that should be considered to those of the type knows;
- $K_{\rho_1} = \{since \rightarrow$ "14.06.2018"$\}$ allowing only relations that have the value *"14.06.2018"* assigned to the property key *since*;
- and $I_{\rho_1} = (null, null)$ and therefore $(m,n) = (1,1)$, which makes this *relation pattern* **rigid**.

Since $K_{\rho_1} \neq \emptyset$, we have to access edge attributes and therefore need to use the edge list table. We check if the label of the edge is knows and if the value assigned to the key *since* is *"14.06.2018"*. The resulting CTE $\rho_{1_{cte}}$ is depicted in Listing 5.70.

Next we inductively translate $\rho_2 = (d_{\rho_2}, a_{\rho_2}, T_{\rho_2}, K_{\rho_2}, I_{\rho_2})$ with

- $d_{\rho_2} = \leftrightarrow$ defining the direction of this *relation pattern* as undirected (or both directions);
- $a_{\rho_2} = l$ assigning the name $l$ to this *relation pattern*;
- $T_{\rho_2} = \{likes\}$, which restricts the considered relations to those of type likes;
- $K_{\rho_2} = \{\}$ not putting any restrictions on properties of the edges;
- and $I_{\rho_2} = (null, null)$ and therefore $(m,n) = (1,1)$ defining the *relation pattern* as rigid.

Because $K_{\rho_2} = \emptyset$ we do not need to access the attributes of the edge and therefore, we can use the adjacency tables. However, the direction $d_{\rho_2} = \leftrightarrow$ is defined as undirected. This means we need to construct a CTE that covers both edge directions. The only remaining restriction is that the relationship has to be of the type likes. The CTE $\rho^{\rightarrow}_{2_{cte}}$ that computes the results for the left-to-right direction is depicted in Listing 5.71. Analogously, the CTE $\rho^{\leftarrow}_{2_{cte}}$ that computes the results for the right-to-left direction can be constructed using the incoming adjacency table.

This finally leads to $\rho_{2_{cte}}$ combining both directions depicted in Listing 5.72.

As the last step, we need to make sure that no edge is used twice in the computed paths. The final resulting CTE is depicted in Listing 5.73 □

```
WITH ρ1cte AS (
  SELECT
    sid AS ρ1source,
      χ1att,
        eid AS ρ1eid, label AS ρ1label, att AS ρ1att,
      χ2att,
    tid AS ρ1target
  FROM χ1cte, edges, χ2cte
  WHERE sid = χ1vid
    AND tid = χ2vid
    AND label ∈ {knows}
    AND attributes(since) = '14.06.2018'
)
```

Listing 5.70: The *first* relationship pattern $\rho_1$.

```
WITH unshred_edges AS (
  SELECT ρ1cte.*,
    UNNEST(...) AS label,
    UNNEST(...) AS eidTmp,
    UNNEST(...) AS tmp
  FROM ρ1cte, outgoing_adjacency
  WHERE vertexid = χ2vid
),
gather_edges AS (
  SELECT ρ1cte.*, array_elements(eidTmp) AS ρ2eid,
    label AS ρ2label, array_elements(tmp) AS ρ2target
  FROM unshred_edges
  WHERE label ∈ {likes}
),
ρ2→cte AS (
  SELECT
    ρ1cte.*,
        ρ2eid, ρ2label,
      χ3att,
    ρ2target
  FROM gather_edges, χ3cte
  WHERE ρ2target = χ3vid
)
```

Listing 5.71: The first direction $\rho^{\rightarrow}_{2_{cte}}$ *second* relationship pattern $\rho_2$.

```
WITH ρ2cte AS (
  SELECT *
  FROM ρ2cte→
    UNION
  SELECT *
  FROM ρ2cte←
)
```

Listing 5.72: The completed second relationship pattern $\rho_2$.

```
WITH match(π,G) AS (
  SELECT *
  FROM ρ2cte
  WHERE ρ1eid ≠ ρ2eid
)
```

Listing 5.73: The final $match(\pi, G)$ CTE.

**Translating *Relationship Patterns* of Variable Length** The translation of *Cypher path patterns* that contain paths of variable length brings an additional requirement to the translation mechanism: Up to now, we have only considered **rigid** patterns, for which we know the number of required JOIN operations beforehand.

For the translation of variable length patterns we use recursive *SQL* queries. To do this, we differentiate between two cases.

First, the *relationship pattern* of variable length is the first *relationship pattern* of the *path pattern*.

Let $\chi_i$ be a node pattern that we already have translated, let $\pi$ be a path pattern and let $\rho = (d, a, T, K, I)$ be a relationship pattern that is part of the *path pattern* $\pi_{complete} = \chi\rho\pi$:

- Since we know that $\rho$ is **not rigid**, we can assume that $I = (m, n)$ with $m, n \in \mathbb{N}_0$ and $m < n$;
- We also assume $d = \rightarrow$ for a simplified presentation. As described before, we can analogously generate the required CTEs for other directions by tracing along the edge list entries in flipped order;
- Let $k \in \{k_1, \ldots, k_o\}$ be all keys for which $K(k)$ is defined;
- and last, we assume that $a \neq null$, otherwise we assign a unique name.

The pseudo *SQL* code in Listing 5.74 depicts the skeleton that we use, if the variable length *relationship pattern* is the first of the path. The base case of the recursion is nearly equal to the CTE of a rigid pattern, except that we need to add a column for the current recursion depth, as well as collecting the already used edge ids.

All results returned by this CTE satisfy the given (part of) the *path pattern*, because

- We assume $a \neq null$. This means $u(a) = list(r_1, \ldots, r_m)$ must hold. This is achieved by defining $u(a) = \rho_1$route;
- $(n, G, u) \models \chi$ follows by construction of $\chi_{cte}$ and $(remainder, G, u) \models \pi$ follows by construction of the remaining *path pattern*;
- $\lambda(r_i) \in T$ holds because of $label \in T$ in both the base, as well as in the recursive case;
- $[\![\sigma(r_i, k) = P(k)]\!]_{G_u} = true$ holds, because of $attributes(k_i) = K(k_i)$ for every $k_i$ that is defined in $K$;
- and $(src(r_i), target(r_i) = (n_i, n_{i+1})$ holds because of $sid = \chi_1$vid and $tid = \chi_2$vid.

Using the recursion *depth* parameter we make sure, that the computed paths have the correct length. Because of the restriction that edges must not be used more than once in a single path (defined in Definition 5.46), we need to track already used edges. We can check this in every step of the recursive step by using the *route* result column.

```
WITH RECURSIVE varpath (ρ1source, χ1att, ρ1label,
                        ρ1att, ρ1target, depth, route) AS (
    (
     SELECT
       sid AS ρ1source,
         χ1att,
           eid AS ρ1eid, label AS ρ1label, att AS ρ1att,
         χ2att,
       tid AS ρ1target,
       1 , ARRAY[eid] AS route
     FROM χ1cte, edges
     WHERE χ1vid = sid
       AND label ∈ T
           AND attributes(k1) = K(k1)
            ... AND attributes(ko) = K(ko)
    )
  UNION ALL
    (
     SELECT
       sid AS ρ1source,
         χ1att,
           eid AS ρ1eid, label AS ρ1label, att AS ρ1att,
         χ2att,
       tid AS ρ1target,
       depth + 1, route || eid
     FROM edges, varpath, χ2cte
     WHERE ρ1target = sid AND tid = χ2vid
       AND label ∈ T
           AND attributes(k1) = K(k1)
            ... AND attributes(ko) = K(ko)
           AND NOT eid = ANY(route)
           AND depth < n
    )
), ρ1cte AS (
  SELECT ρ1source, χ1att,
    null, null, ρ1route, χ2att, ρ1target
  FROM varpath
  WHERE depth >= m
)
```

Listing 5.74: The base case for a variable *relationship pattern*.

```
WITH RECURSIVE varpath (ρ_i source, χ_i att, ρ_i label,
                        ρ_i att, ρ_i target, depth, route) AS (
    (
     SELECT
       sid AS ρ_i source,
         χ_i att,
           eid AS ρ_i eid, label AS ρ_i label, att AS ρ_i att,
         χ_{i+1} att,
       tid AS ρ_i target,
       1 , ARRAY[eid] AS route
     FROM ρ_{i-1}, edges
     WHERE ρ_{i-1} target = sid
       AND label ∈ T
           AND attributes(k_1) = K(k_1)
            ... AND attributes(k_o) = K(k_o)
    )
  UNION ALL
    (
     SELECT
       sid AS ρ_i source,
         χ_i att,
           eid AS ρ_i eid, label AS ρ_i label, att AS ρ_i att,
         χ_{i+1} att,
       tid AS ρ_i target,
           depth + 1, route || eid
     FROM edges, varpath, χ_{i+1}
     WHERE ρ_{i-1} target = sid AND tid = χ_{i+1} vid
       AND label ∈ T
           AND attributes(k_1) = K(k_1)
            ... AND attributes(k_o) = K(k_o)
           AND NOT eid = ANY(route)
           AND depth < n
    )
), ρ_{i_cte} AS (
  SELECT ρ_i source, χ_i att,
    null, null, ρ_i route, ρ_i target, χ_{i+1} att
  FROM varpath
  WHERE depth >= m
)
```

Listing 5.75: The inductive case for a variable *relationship pattern*.

Second, if the *relationship pattern* is not the first of the path, we can analogously construct a CTE by replacing $\chi_{1_{cte}}$ with the already constructed relation CTE $\rho_{i-1}$. This CTE skeleton is depicted in Listing 5.75.

After we have now defined the translation of arbitrary *relationship patterns*, we can combine the building blocks constructed by the translation mechanism for rigid *path patterns* and the building blocks that are generated by the mechanism for *relationship patterns* of variable length. This is possible, because as before we do not restrict the input CTE $\rho_{i-1}$ to be of the same *relationship pattern* type.

We create the final CTE for the match operator by selecting the results of the last CTE we created for the match clause. Since we always use the previously built CTE as input for the next CTE, it contains all necessary results. Finally, we have to make sure that every edge id does only occur once in every path. To this end, we can create a suitable WHERE clause.

The skeleton for the final $match(\pi, G)$ CTE is depicted in Listing 5.76. As before, we have to make sure that no edge is used more than once in a single result path. Here we need to apply the different cases depicted in Listing 5.76 depending on what types of *relationship patterns* have been used.

**Example 5.77**
Let us revisit the example introduced in Example 5.49 using the MATCH clause in Listing 5.50. Therefore, we are considering the *path pattern* $\pi = \chi_1\rho_1\chi_2\rho_2\chi_3$. The first *relationship pattern* $\rho_1 = (\rightarrow, a_{\rho_1}, T_{\rho_1}, K_{\rho_1}, I_{\rho_1})$ with $I_{\rho_1} = (1,2)$ is the only *relationship pattern* in the given *path pattern* that is not rigid. Therefore, let us assume that we have already constructed $\chi_{1_{cte}}, \chi_{2_{cte}}$ and $\chi_{3_{cte}}$.

We use the skeleton for the base case described above using the already translated *node pattern* $\chi_{1_{cte}}$ as the start node. Since we do not have any restrictions on attributes, we do not need to add constraints for $K_{\rho_1}$. We have $T_{\rho_1} = \{knows, likes\}$, therefore we add constraints for this and also include the path length restrictions for $I_{\rho_1} = (1,2)$. The result CTE $\rho_{1_{cte}}$ for the construction is depicted in Listing 5.78.

$$
\begin{array}{l}
\textbf{WITH } match(\pi, G) \textbf{ AS } ( \\
\quad \textbf{SELECT } \star \\
\quad \textbf{FROM } \rho_{k_{cte}} \\
\quad \textbf{WHERE } \forall i, j \in \{1, \ldots, k\}, i \neq j : \begin{cases} (\rho_i eid \neq \rho_j eid) & \texttt{,if } \rho_i, \rho_j \texttt{ rigid,} \\ (\rho_i eid \notin \rho_j route) & \texttt{,if only } \rho_i \texttt{ rigid,} \\ (\rho_j eid \notin \rho_i route) & \texttt{,if only } \rho_j \texttt{ rigid,} \\ (\rho_i route \cap \rho_j route = \emptyset) & \texttt{,if } \rho_i, \rho_j \texttt{ variable.} \end{cases} \\
)
\end{array}
$$

Listing 5.76: The complete CTE to compute an arbitrary $match(\pi, G)$.

```
WITH RECURSIVE varpath (ρ1source, χ1att, ρ1label,
                        ρ1att, ρ1target, depth, route) AS (
    (
     SELECT
       sid AS ρ1source,
         χ1att,
           eid AS ρ1eid, label AS ρ1label, att AS ρ1att,
         χ2att,
       tid AS ρ1target,
       1 , ARRAY[eid] AS route
     FROM χ1cte, edges
     WHERE χ1vid = sid
       AND label ∈ {knows, likes}
    )
  UNION ALL
    (
     SELECT
       sid AS ρ1source,
         χ1att,
           eid AS ρ1eid, label AS ρ1label, att AS ρ1att,
         χ2att,
       tid AS ρ1target,
           depth + 1, route || eid
     FROM edges, varpath, χ2cte
     WHERE ρ1target = sid AND tid = χ2vid
       AND label ∈ {knows, likes}
           AND NOT eid = ANY(route)
           AND depth < 2
    )
), ρ1cte AS (
  SELECT ρ1source, χ1att,
    null, null, ρ1route, χ2att, ρ1target
  FROM varpath
  WHERE depth >= 1
)
```

Listing 5.78: Example base case for the variable *relationship pattern* $\rho_1$.

```
WITH match(π,G) AS (
  SELECT *
  FROM ρ2cte
  WHERE ρ2eid ∉ ρ1route
)
```

Listing 5.79: Finished translated match relation for $\pi$.

We translate the rigid *relationship pattern* $\rho_2$ as we have shown in Example 5.69 using $\rho_{1_{cte}}$ as the input CTE and retrieve $\rho_{2_{cte}}$. After we have done this, we have constructed all parts of the *path pattern* $\pi$ and can now construct the CTE $match(\pi, G)$ that computes all valid results for $\pi$ and the graph $G$. This final CTE is depicted in Listing 5.79. □

**Mapping *Cypher* Query Operators** We will now describe the mapping of *Cypher* queries to *SQL* queries for **RATG** based on the syntax described in Section 5.2.1.3. The general concept is depicted in Figure 5.80:

We translate the MATCH clause into a CTE as we have described in Section 5.2.2.1. For the WHERE clause we have to differentiate between the use of WHERE as an *SQL* selection and the use of HAVING for the GROUP BY operator. Finally, the RETURN clause can directly be translated to the SELECT clause of *SQL*.

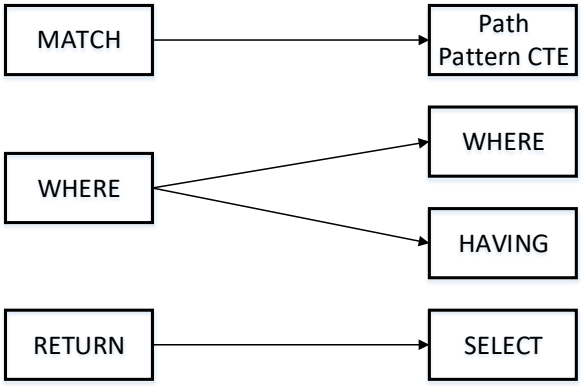



Figure 5.80.: Mapping of basic *Cypher* clauses.

Let us now have a closer look at the translation based on the *Cypher* syntax (see Section 5.2.1.3).

#### 5.2.2.2. Constructing SQL Queries from *Cypher*

We have already described our translation of MATCH clauses in the previous section. Therefore, we now take a look at the translation of the next building block of *Cypher*: The **Reading Clause**.

We will need to remember, which of the CTE represents a specific named node or relationship. We describe this for the node or relationship $a$ in the CTE $SubQuery_{cte}$ by using the function $\tau(a, SubQuery_{cte})$.

In our database schema, the complete representation of a node is the combination of the columns VID and *attributes*, while the complete representation of an edge consists of the edge's id EID, its source id SID, the target id TID, the label, and the set of attributes. In *Cypher* we can simply return a whole node or relationship using the variable that is used within the query.

To shorten the representation of the SELECT clauses in the following sections, we define $\mathrm{T}(a, SubQuery_{cte})$ (with $a$ either a name for a node or a relationship) as the function that returns the complete representation of a node or relationship.

For the sake of simplicity, we also assume that we always use the complete node or relationship data for RETURN and WITH clauses. If this is not the case, the selected columns in the *SQL* queries can easily be adapted by projecting the results to the desired columns.

**Reading Clause** As we know from Figure 5.26, a **Reading Clause** can consist of (OPTIONAL) MATCH clauses, possibly followed by a WHERE clause. Let $\pi_{1_{cte}}, \ldots, \pi_{n_{cte}}$ be the already translated MATCH clauses.

We can now construct a CTE that computes the results for a given **Reading Clause**. We do this by using all CTEs that we constructed to represent the different MATCH clauses contained in the current **Reading Clause** as source tables. Then we make sure to only return every variable (defined in the MATCH clauses) once to avoid redundant data flow. Additionally, we have to make sure that variables that have the same name contain the same values (or are bound to the same instances). To this end, we create a WHERE clause that makes sure of that.

The skeleton we use to translate the **Reading Clause** is depicted in Listing 5.81. We create another CTE in which we perform a JOIN using all common free variables of the different *path patterns*. Those JOIN operations are necessary, because in *Cypher* (just like in *SPARQL*) variables that have the same name assigned are implicitly joined.

The WHERE conditions of the **Reading Clause** can directly be applied using the correct *SQL* syntax, since the semantics of *SQL* and *Cypher* do not differ here. For the previously mentioned listing, we use $WHERE_{ReadingClause}$ to describe the set of conditions defined in the **ReadingClause**.

Note that if a MATCH clause is marked as OPTIONAL we perform an OUTER JOIN instead of a normal join.

```
WITH ReadingClause_cte AS {
  SELECT {T(a, π1)| such that a ∈ {⋂_{i∈{1,...,n}} free(πi)}},
    {T(a, πi)| such that i ∈ {1,...,n}, a ∈ {free(πi) \ ⋃_{j∈{1,...,n}, i≠j} free(πj)}}
  FROM π1_cte, ..., πn_cte
  WHERE ∀i, j ∈ {1,...,n}, i < j:
    ∀a ∈ (free(πi) ∩ free(πj)) : { τ(a, πi_cte)id = τ(a, πj_cte)id,
                                     if a part of a node pattern
                                 { τ(a, πi_cte)eid = τ(a, πj_cte)eid,
                                     if a part of a relationship pattern
    AND WHERE_ReadingClause
}
```

Listing 5.81: Translation skeleton for a **Reading Clause**.

**Example 5.82**
Let us reconsider the example from Section 5.2.1.3. The corresponding *Cypher* fragment is depicted in Listing 5.83.

Let $\pi_{cte}$ be the CTE that we have already translated from the MATCH clause with $\chi_{1_{cte}}$ the CTE representing $p1$, $\chi_{2_{cte}}$ the CTE representing $p$, and $\rho_{1_{cte}}$ the CTE representing the edge.

```
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WHERE p1.name = "Yamamoto"
```

Listing 5.83: Example *Cypher* **Reading Clause**.

Then we construct the CTE depicted in Listing 5.84, in which we only have to set a condition for $p1.name$. In our translation, we can access this node using the naming convention used for $\chi_{1_{cte}}$.

```
WITH ReadingClauseExample_cte AS {
  SELECT *
  FROM π1_cte
  WHERE χ1att(name) = "Yamamoto"
}
```

Listing 5.84: Example Translation of a **Reading Clause**.

□

**Single Part Query** From Figure 5.23 we know that a **Single Part Query** (not considering update, deletion or function call queries) only consists of a **Reading Clause** that is completed with a RETURN clause.

Let $a_1, \ldots, a_i$ be all elements of the RETURN clause of the *Cypher* query. By remembering which *node pattern* (respectively *relationship pattern*) represents the different named nodes (and respectively relationships), we can easily construct the corresponding SELECT clause.

```
WITH SinglePartQuery_cte AS (
  SELECT {T(a, ReadingClause_cte) | such that a ∈ RETURN_ReadingClause_cte}
  FROM ReadingClause_cte
)
```

Listing 5.85: Translation skeleton for a **Single Part Query**.

Since the **Single Part Query** only further restricts the set of returned variables, we omit an example at this point.

**Multi Part Query** As seen in Figure 5.24, a **Multi Part Query** can combine several **Reading Clauses** by using WITH and ends with a **Single Part Query**. We can observe that the WITH statement acts exactly like the RETURN statement of a CTE in *SQL*.

```
WITH MultiPartQuery_1cte AS (
  SELECT {T(a, ReadingClause_1cte) | such that a ∈ WITH_ReadingClause_1cte}
  FROM ReadingClause_1cte
)
```

Listing 5.86: Translation skeleton for the **Multi Part Query** base case.

As we have done several times before, we inductively construct the CTE representing this **Multi Part Query**. Therefore, let us assume that we have already translated the corresponding **Reading Clauses**: For $i \in \{1, \ldots, n\}$ let $ReadingClause_{i_{cte}}$ be the **Reading Clause** used in the i'th part of the **Multi Part Query**. We describe the **set** of returned elements of a given part $MultiPartQuery_{i_{cte}}$ of the **Multi Part Query** as $WITH_{MultiPartQuery_{i_{cte}}}$.

```
WITH MulitPartQuery_ncte AS (
  SELECT {T(a, ReadingClause_ncte) | such that a ∈ WITH_ReadingClause_ncte}
  FROM MultiPartQuery_n-1cte, ReadingClause_ncte
  WHERE ∀a ∈ (WITH_ReadingClause_ncte ∩ WITH_MultiPartQuery_n-1) :
    { τ(a, MultiPartQuery_n-1cte)id = τ(a, ReadingClause_ncte)id, if a is a node
    { τ(a, MultiPartQuery_n-1cte)eid = τ(a, ReadingClause_ncte)eid, if a is an edge
```

Listing 5.87: Translation skeleton for the **Multi Part Query** inductive case.

The first **Reading Clause**/WITH combination of the **Multi Part Query** is translated the same way a **Single Part Query** is translated. We create a CTE that represents this part by projecting the results to the set of variables defined in the corresponding WITH statement $WITH_{ReadingClause_{1_{cte}}}$. The skeleton for this is depicted in Listing 5.86.

```
WITH MulitPartQuery_final_cte AS (
  SELECT {T(a, MultiPartQuery_n_cte) | such that a ∈ RETURN}
  FROM MultiPartQuery_n_cte
)
```

Listing 5.88: Translation skeleton for the **Multi Part Query** final case.

For the inductive case (depicted in Listing 5.87) we create a join between the already translated $MultiPartQuery_{n-1_{cte}}$ and the current **Reading Clause** $ReadingClause_{cte}$. To do this, we perform an equi-JOIN on any id for which the same named variable occurs in both, $MultiPartQuery_{n-1_{cte}}$ and $ReadingClause_{cte}$. We have to do this for the same reason for which we had to perform the joins in the **Reading Clause** before. Variables that are named identically represent the same vertex, and therefore describe an implicit *JOIN*. We can achieve this using a mapping between the names of the variables and the names of the node and relationship CTEs that are available in the respective **Multi Part Query** and **Reading Clause** CTEs. To finish the inductive construction, we project the results in such a way that we carry all vertice and edge data that is contained in the result.

Finally, we complete the translation of the **Mulit Part Query** by applying the skeleton depicted in Listing 5.88, which uses the last **Reading Clause** in combination with its corresponding RETURN clause (this combination also represents a **Single Part Query**) to project the results of the previous temporary view to the correct set of returned columns.

**Example 5.89**
Consider the example in Section 5.2.1.3 again. The corresponding query is depicted in Listing 5.90. Let $ReadingClause_{1_{cte}}$ be the CTE representing the first MATCH of the *Cypher* fragment, which contains $\pi_{1_{cte}}$. In turn, $\pi_{1_{cte}}$ contains $\chi_{1_{cte}}$ that represents *p1*.

```
MATCH (p1:Person{name:"Yamamoto"})
WITH p1
MATCH (p1:Person) -[:KNOWS]-> (p:Person)
WITH p1, COUNT(p) AS numberOfFriends
WHERE numberOfFriends > 1
RETURN p1
```

Listing 5.90: Example **Multi Part Query**.

Let $ReadingClause_{2_{cte}}$ be the CTE that represents the second **Reading Clause**, including $\pi_{2_{cte}}$. $\pi_{2_{cte}}$ contains $\chi_{2_{cte}}$ (that also represents *p1*), $\chi_{3_{cte}}$ (representing *p*) and $\rho_{1_{cte}}$ which represents the relationship of type KNOWS. In addition, an aggregation function call (COUNT(*p*)) is used in this example.

Note that the given example *Cypher* query contains an aggregation operator. This aggregation works exactly the same as in *SQL*, except that *Cypher* does not explicitly define the GROUP BY clause. Instead, exactly the variables that are returned by the WITH clause are part of the GROUP BY.

```
WITH MulitPartQueryExample1cte AS (
  SELECT χ1id, χ1att
  FROM ReadingClause1cte
), MultiPartQueryExample2cte AS (
  SELECT χ1id, χ1att, COUNT(χ1vid) AS numberOfFriends
  FROM MulitPartQuery1cte, ReadingClause2cte
  WHERE χ1id = χ2id
  GROUP BY χ1vid, χ1att
  HAVING COUNT(χ1vid) > 1
), MultiPartQueryExamplefinalcte AS (
  SELECT χ1id, χ1att
  FROM MultiPartQueryExample2cte
)
```

Listing 5.91: An Example for the Translation of a **Multi Part Query**.

This leads to the resulting temporary view depicted in Listing 5.91. $MultiPartQueryExample_{1_{cte}}$ represents the first MATCH/WITH pair, and therefore also the base case for our inductive construction. We already have translated $ReadingClause_1$ into $ReadingClause_{1_{cte}}$, and therefore ready for use. Hence, we only need to project on the variables defined by the WITH phrase. It only defines the vertex *p1* to be returned, therefore we SELECT the complete representation of *p1* defined by $\chi_1$id and $\chi_1$att. These are available through $\chi_{1_{cte}}$, which in turn is part of $ReadingClause_{1_{cte}}$.

Next, we apply the inductive case skeleton to the second **Reading Clause**/WITH pair. We also have already translated the **Reading Clause** into the temporary view $ReadingClause_{2_{cte}}$. To this end, we JOIN $ReadingClause_{2_{cte}}$ and $MultiPartQueryExample_{1_{cte}}$ on all variables the corresponding WITH statements have in common. In this case, the only common variable is *p1*, which is represented by $\chi_{1_{cte}}$ in $MultiPartQueryExample_{1_{cte}}$ and by $\chi_{2_{cte}}$ in $ReadingClause_{2_{cte}}$.

Because the WITH clause that we are currently translating contains the aggregation operator COUNT, we need to apply the corresponding GROUP BY and COUNT operators of *SQL*. The attributes we GROUP BY are defined by the WITH clause – in this case *p1*. We also have to translate the condition $numberOfFriends > 1$, which we can do by using the HAVING clause of *SQL*. Finally, we set the variables defined by $WITH_{ReadingClause_{2_{cte}}}$ as the returned values.

For the final temporary view $MultiPartQueryExample_{final_{cte}}$, we simply project on the set of variables defined by the RETURN clause, which is *p1* and is represented by $\chi_{1_{cte}}$. Because the complete vertex is represented by two columns of our schema, we return both $\chi_1$id and $\chi_1$att. □

**Single Query** The translation of the **Single Query** is simply the choice between translating a **Multi Part Query** or a **Single Part Query**. Therefore, we can query the results of the input CTE. The complete skeleton for the **Single Query** is depicted in Listing 5.92.

```
SELECT *
FROM MultiPartQuery_{final_cte}
```

Listing 5.92: The Translation Skeleton for a **Single Query**.

**Cypher Query** Finally, we can translate the complete *Cypher* Query. From Figure 5.19 we know that a *Cypher* query is the UNION of an arbitrary number of **Single Queries**. Since the UNION operator of *Cypher* and *SQL* have the same conditions we can construct a simple UNION between the CTEs that represent the **Single Queries** of the *Cypher* query. Let $SingleQuery_{1_{cte}}, \ldots, SingleQuery_{n_{cte}}$ be the **Single Queries** of the given *Cypher* query. Then the resulting *SQL* query is depicted in Listing 5.93.

```
SELECT *
FROM SingleQuery_{1_cte}
  UNION
  ...
  UNION
SELECT *
FROM SingleQuery_{n-1_cte}
  UNION
SELECT *
FROM SingleQuery_{n_cte}
```

Listing 5.93: Translation skeleton for the final **Translated Cypher Query**.

**Remark 5.6**
Please note that the translation procedure previously described in this chapter does not include efficiency considerations. Obviously, the resulting *SQL* queries offer ample opportunity for optimization. Some of those optimizations we will describe in the next Section. □

#### 5.2.2.3. Optimizing the Resulting Queries

The mechanism we have described in Sections 5.2.2.1 and 5.2.2.2 did not include any performance considerations. As we have described before, we make massive use of CTEs. Up to version 11 **PostgreSQL**'s query optimizer was not capable of optimizing queries across different CTEs. This means the optimizer would try to make each CTE itself efficient, but for example, would not perform selection pushing as known from *Relational Algebra*. [Kem15]

From version 12 onward, **PostgreSQL**'s query optimizer is able to optimize across CTEs. Unfortunately, a short preliminary evaluation has shown that the optimizer is overtaxed with the optimization of the complicated queries resulting from our translation mechanism. In some cases, this can even lead to serious performance degradation. This results in some queries that should terminate in a few seconds not terminating at all. Therefore, we need to apply our own query optimization.

To achieve this, we have to consider two questions: First, when do we use adjacency lists or the edge lists. Second, in what order do we create the CTEs representing the egdes of the *Cypher* patterns.

In order to achieve good performance using our translation mechanism, we optimize the resulting *SQL* queries using standard optimization techniques known from *Relational Algebra* (see e.g. [Kem15] and *System R* [Ast+76]).

We will now give a short overview of optimizations we apply to the *Cypher* to *SQL* query translation mechanism. A detailed description of the optimizations we apply can be found in [Goj21].

**Early Projection** The first optimization we apply is early projection. The translation mechanism described in Section 5.2.2.1 does not consider, which attributes or labels are required to return the results for a given query. While this is not necessary to compute the correct results for a given query, it can massively impact the size of intermediate results.

Let us revisit the example from Section 5.2.2.1 depicted in Listing 5.94:

```
MATCH (p:Person) -[:KNOWS*2]- (m:Person) -[:LIKES]-> (o:Post)
RETURN p.firstName, o.id
```

Listing 5.94: Rigid *Cypher* Query Example.

Using the translation mechanism exactly as described in Section 5.2.2.1, the resulting *SQL* query would look like the query depicted in Listing 5.95. All attributes that belong to the nodes that are matched to the first *node pattern* are carried to the end of the query, at which point the result set is projected to the desired columns.

We can apply early projection, since we know that we do not need any attributes from Person type nodes other than the *firstName*. The resulting query is depicted in Listing 5.96.

```
WITH χ1cte AS {
    SELECT p1.vid AS χ1vid, p1.attributes AS χ1att
    FROM vertices p1
    WHERE p1.attributes(type) = Person
},
...
,
SELECT χ1att(firstName), ...
...
```

Listing 5.95: SQL Query Without Early Projection.

```
WITH χ1cte AS {
    SELECT p1.vid AS χ1vid, p1.attributes(firstName) AS χ1firstName
    FROM vertices p1
    WHERE p1.attributes(type) = Person
},
...
,
SELECT χ1firstName, ...
...
```

Listing 5.96: SQL Query Without Early Projection.

Here we only keep the *firstName* attribute when matching for suitable Person vertices.

In addition to early projection, we can also apply the well known optimization concept **Early Selection** (also known as **Selection Pushing**) to the resulting queries. For further details please see [Goj21].

**Choosing Target Tables and Edge Order Optimization** Choosing the correct order in which to construct the CTEs representing the *Cypher* query is the most impactful optimization we can apply. Our tests showed improvements of up to several orders of magnitude.

In order to find a good edge order, we applied a heuristic dynamic programming approach adapted from the-well known join-order-optimization problem [SMK97]. First results show great promise but exceed the frame of this work. For a detailed look at our findings see [Goj21].

# 6. Performance Evaluation

For this part of our work, the main goal was to compare the read-query performance of our adaptation with the *SQLGraph* schema. In addition, we were interested in the performance our approach achieves with regard to update queries. To this end, we used the *Interactive Workload* of the *LDBC-SNB* [Bon13; Erl+15; Szá+18].

The evaluation of the performance of our approach in regard to the IFC building data use case can be found in Chapter 10.

## 6.1. The Linked Data Council - Social Network Benchmark: Interactive Workload

In order to fully evaluate the performance of the presented approach we required different data set sizes: First, we needed to evaluate our solution on data sets that fit into main memory of the system. Second, we also wanted to evaluate the performance on data sets that are much too big to fit into main memory.

The LDBC-SNB provides a data set generator that can generate data sets of arbitrary size. We chose the LDBC-SNB Interactive Workload, because it fulfills the following criteria:

- Many of the queries have paths of fixed length as their main pattern and use edges of known label. This criterion is desired, because preliminary evaluations (see [Kor17]) have shown, that in other cases the use of the attribute table performs better than the adjacency tables. The Interactive Workload offers a variety of different queries that contain both: paths of fixed length and paths of variable length. In some queries both types of paths occur.
- All combinations of adjacency tables and the edge table are required. Therefore, the set of queries requires the use of incoming, outgoing and undirected edges to cover all combinations of the redundant edge storage. The chosen benchmark contains queries with edges in different directions, as well as undirected edges.
- Different lengths of paths are contained in the set of queries to determine, if the difference in path length makes one of the two schema versions preferable. The Interactive Workload contains queries that only query a single vertex as well as queries with paths of considerable length and optional parts.

- Because [Sun+15] qualifies this type of query as a potential bottleneck of the original approach, queries with big intermediate result sets are part of our chosen query set. Part of the complex queries of the workload are designed with huge intermediate result sets in mind.
- The set of queries contains updates. This should include edges between already existing vertices and also queries that insert edges to new vertices. The workload's update queries contain both of those cases.

## 6.2. The Methodology

First, we confirmed the requirement of redundant representation of edges. To this end, we defined path queries with different fixed lengths and also path queries of variable lengths. After we confirmed the necessity of the edge tables, we evaluated our approach with the LDBC-SNB (version 0.3.1).

The parameters required by the queries are produced by the LDBC-SNB data set generator in advance. Therefore, exactly the same parameters were used for both approaches.

To make the performance of the two schemas as comparable as possible, we first implemented all queries for **RATG**. Then we replaced only the necessary subqueries that concern the differences between the schemas, changing as little of the query as possible. We confirmed the correctness of our query implementations by comparing the results with results returned by a reference implementation provided for **Neo4j**, of which correctness has been validated.[1]

Since even the insertion of a single edge requires updates on several tables, we implemented the update queries using **PostgreSQL** functions (the **PostgreSQL** equivalent to stored procedures). This way, several round trip times between the client and the server can be omitted. All update queries are based on a basic *insertEdge()*-function that handles all the insertions in the edge attribute table and the adjacency tables. This still leaves room for future improvement. If a query inserts multiple edges for a single vertex the same array has to be updated multiple times. Therefore, query specific functions, that do not need to perform multiple updates on the same adjacency list should lead to further improvement.

We ran the benchmark on the same hardware using the same data set. The evaluation was conducted on a dedicated server with the following specifications.

- Two Intel Xeon 2.6GHz CPUs (in total 8 cores);
- 96 GB main memory;
- A 6 SSD RAID-0 (other RAID configurations would not have left enough storage capacity for the biggest data set);
- Running 64-bit Ubuntu (version 18.04 LTS).

[1] https://github.com/PlatformLab/ldbc-snb-impls/tree/master/snb-interactive-neo4j

We used **PostgreSQL 13** and **Neo4j 3.4** on the aforementioned server, while we ran the client program of the benchmark on a standard desktop PC that connected to the database over LAN with a 1 Gbit/s connection and both machines connected to the same switch. We tuned both databases to the best of our abilities by using the evaluation framework described in Section 6.2.1.

All queries that were performed for performance evaluation purposes were preceded by several hundred warm-up queries as advised by Dominguez [Dom+10].

### 6.2.1. Performance Evaluation Framework

The efficiency of a relational database can vary greatly depending on various tuning parameters, data set size and many other variables. In order to achieve conclusive results, we wanted our benchmark executions to be easily repeatable and therefore also reproducible. Hereby, we are able to compare different values and settings for the numerous parameters and find a suitable configuration of the database.

Thus, we designed an evaluation framework that automates most of the benchmark execution process. We then used this framework to repeatedly run benchmark executions to find the best tuning parameters and application-specific indexes for the LDBC-SNB. The main components and tasks of this framework are depicted in Figure 6.1.

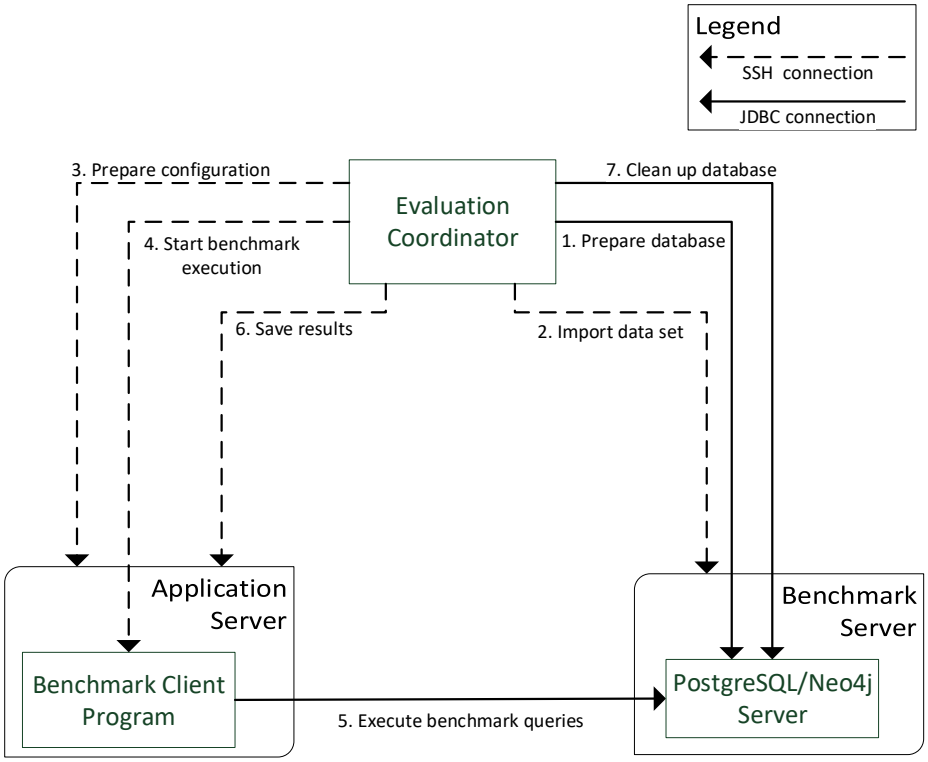



Figure 6.1.: Components and tasks of the evaluation framework.

Our framework contains three components that can freely be distributed across different machines.

**Evaluation Coordinator:** The evaluation coordinator is the program that coordinates the whole benchmark run cycle. To this end, it creates Secure Shell (SSH) and JDBC connections to the different machines used in the evaluation process. It starts the database services, triggers the different tools required to set up a run, runs the benchmark program, and saves results. We used a standard desktop computer running **Ubuntu 18.04**. The computer was connected to the other machines through a 1 Gbit/s LAN connection and all computers were connected to the same switch. Thus, network delay was minimized.

**Application Server:** The application server is the machine that executes the benchmark program. For this role we used the same computer as we used for the evaluation coordinator. Since the evaluation coordinator is inactive while the benchmark client program is running and starts its next task only after the benchmark run has finished, this does not hinder performance.

**Benchmark Server:** The benchmark server is the machine that runs the database management systems. This means the server ran both, **PostgreSQL** and **Neo4j**. Depending on which benchmark execution is chosen, the evaluation coordinator makes sure that only the database system under test is active while the other service is stopped.

In this thesis we use the term *benchmark run* for the *complete execution* of the benchmark client program for a *single data set size* on a *single database system*. For example to compare the results of **RATG** , the **SQLGraph**, and **Neo4j**, we need to execute three benchmark runs. To perform a single run, the evaluation framework performs the following tasks in order (see Figure 6.1):

- The first list of tasks initializes the database and configures the benchmark client program:

    1. The coordinator connects to the server machine and starts the database system that is going to be evaluated. It also makes sure that the other database system is stopped. This guarantees that no other unnecessary processes influence the measured performance results.

    2. Afterwards, the coordinator imports the current data set into the database system. The data set is freshly imported into the database for every execution, so that it is the only data that is present in the system. Before the coordinator proceeds to the next step, it makes sure that necessary database maintenance tasks have been performed. These tasks for example comprise the analysis of the stored data for index selection during query optimization. Since we import huge data sets, this significantly influences performance.

    3. After the data has been imported into the database, the coordinator connects to the client machine and prepares the benchmark configuration. This step is

necessary, because some parameters that are required for a benchmark execution (for example, parameters that are used for queries) are generated with the data set and therefore depend on the data set size.

- After the systems have been initialized, the benchmark run is started:
  4. The coordinator connects to the client machine and starts the LDBC-SNB run for the current data set with the prepared configuration.
  5. The benchmark client program connects to the database and performs the benchmark execution for the current data set. The results of this execution are stored in a temporary folder.
- Next the current benchmark run is finalized:
  6. The results of the current run are moved from the temporary to a persistent folder.
  7. Finally, the evaluation coordinator connects to the database machine, deletes the data set from the database, and stops the database service. Now both, the database server and the client machine are ready for the next run.

The evaluation framework can be configured to automatically perform the benchmark runs for different data set sizes and database systems.

We used this framework to automatically execute runs for the scale factors 1, 3, 10, 30 and 100 on **RATG**, the original **SQLGraph** and **Neo4j**. We tuned all systems under test to the best of our abilities and then performed final benchmark runs on the best configurations we found. Note that we did not evaluate complex query 14, because the given implementation of **Neo4j** relied on a specific internal implementation of a shortest path algorithm. We do not intend to compare *SQL* queries to internal database functions. Therefore, we deactivated this query for the given benchmark executions. The results can be found in Section 6.3.

### 6.2.2. Application Specific Indexes

Using the aforementioned evaluation framework, we also analyzed the queries in order to create application-specific indices as described in Section 3.2.1. These types of index are inevitable in order to achieve high performance for a given application.

To obtain the best performance we could achieve for the LDBC-SNB benchmark, we added the following application-specific indices to the *vertex table:*

***Composite Index* on *ID* and *Type*** Most of the LDBC-SNB queries start at a given vertex. Most of the time, this vertex is queried by its identifier in the data set. This identifier consists of the ID attribute combined with the *Type* attribute. In order to speed up the process of finding the starting point for the query, we add an index on the complex identifier using a B-Tree. Note that this *ID* attribute originates in the data set and is not the vertex's VID.

***Composite Index* on VID and *Type*** Some of the queries contain neighborhood queries that only search for a certain vertex *Type* of neighbors. Since this information is not stored in the *adjacency list* tables that we use to achieve this kind of query, a JOIN with the *vertex table* is required. Creating a composite B-Tree index on the *VID* that is required to join the *vertex table* to the targets of the *adjacency list* and the *Type* attribute enables the DBMS to check the *Type* of the target vertex using an index-only-look-up. Therefore, the vertex data does not need to be loaded, which leads to a speed-up of this type of query.

We also added the following application specific-indexes to the *edge list* table. Since the *edge list* table is mainly used for recursive queries, these indices essentially improved the performance of queries that use graph patterns of variable length:

***Composite Index* on SID and Label** In order to speed up queries that use graph patterns of variable length, we add a composite index on the source id SID and Label. These queries result in complex recursive queries that use the *edge list* and nearly all of these queries only use specific labels.

***Composite Index* on TID and Label** Since some of the queries in the LDBC-SNB use graph patterns of variable length with undirected edges, we also need to be able to efficiently traverse along edges in the opposite direction, while some of the queries require to trace along the edge from target to source. Therefore, we also add a combined index on TID and Label.

***Composite Index* on SID, TID and Label** We add this composite index to speed up some queries that need to check for the existence of an edge between two vertices that have already been added to the intermediate result set. Consider Listing 6.2 for an example of such an edge. Assume that the DBMS starts at Person vertex *p1*, follows the edges to the intermediate Person vertices and from there finds possibly relevant Person vertices *p3*. Now the existence of the knows edge *r* between *p1* and *p3* must be verified. Using

this index, the DBMS can check the existence of this edge using an index-only-lookup without the need to load the table itself.

***Composite Index* on TID, SID and Label** We add this index analogously to the previously described composite index.

We did not find any additional indexes for the *outgoing adjacency* table and the *incoming adjacency* table that had a positive effect on the benchmark execution performance.

```
MATCH (p1:Person) -[:KNOWS]- (:Person) -[:KNOWS]- (p3:Person)
MATCH (p1:Person) -[r:KNOWS]- (p3:Person)
RETURN p3
```

Listing 6.2: Example query that can use an index-only-lookup.

We verified that all indexes we added are used frequently during benchmark executions using **PostgreSQL**'s internal statistics collection functionality.

### 6.2.3. Preliminary Evaluation: Redundant Edge Data

Before we started the actual performance evaluation of our schema, we first conducted a preliminary evaluation to verify the need for the redundant storage of the edge data. This was claimed to be necessary by [Bor+13]. To this end, Kornev [Kor17] conducted an extensive evaluation that examined when to use the *edge list* table in favor of the *adjacency lists*. We found a general tendency for queries to perform significantly better with the use of adjacency tables, whenever the queried path is of fixed length. While this is true for paths of fixed length, the opposite holds for paths of variable length. This type of query requires recursive SQL queries. Kornev's work has shown that recursive queries using the edge attributes table outperform any recursive query that uses our adjacency tables. Therefore, our approach combines the use of adjacency tables with an edge table.

## 6.3. Query Performance

The following sections present the results we achieved using the previously described evaluation framework and the LDBC-SNB. Results for query performance are depicted in box plot diagrams. If not stated differently, the upper bound of the box shows the 75th percentile, the line within the box shows the 50th percentile and the lower bound of the box shows the 25th percentile, while the whiskers show the maximal and minimal execution time respectively.

We will first take a look at the overall throughput of the different approaches when evaluated with the LDBC-SNB. Afterwards, we will take a closer look at the performance the approaches offer in accordance to the different types of queries the LDBC-SNB provides.

### 6.3.1. Overall Throughput and Query Performance

Since we optimized our schema for read-only queries and heavy workloads, we expected update query performance to diminish in comparison to the approach of Sun et al. [Sun+15]. Therefore, we analyzed query execution times and found that the management of huge arrays stored in *PostgreSQL* columns can lead to performance limitations. We address this problem by introducing a new configuration parameter to limit the amount of edges stored in each array. If an array is full, a new tuple for the vertex is introduced and the additional edges are stored there.

Our assumption was that the database system can manage smaller arrays more efficiently and that this will mainly improve update-query performance. Nevertheless, the results we obtained using the LDBC-SNB show that this optimization barely influences update-query performance. On the other hand, it greatly improved read-query execution for several queries.

Again, this led to a significant improvement of overall throughput for benchmark executions. The achieved throughput is depicted in Figure 6.3 and is up to nearly four times higher than before the array element limit optimization.

The increasing throughput performance of the runs with scale factor 10 compared to the throughput of runs with scale factor 3 can be explained as follows: The workload of the *LDBC-SNB* is generated through the use of a model that emulates a realistic social network. This also means that networks that are used by more users also generate more input queries.

The throughput achieved for the scale factor 3 workload is the highest achievable throughput for this scale factor. Since the workload for scale factor 10 simulates more concurrent users and therefore also generates more queries per second, the highest achievable throughput for scale factor 10 is much higher than the throughput that can be achieved for scale factor 3.

Our findings show, that **RATG** performs significantly better for the LDBC-SNB in all scale factors compared to **SQLGraph**. While we achieve similar results to **Neo4j** for scale factor 3 and 10, **Neo4j** performs better for higher scale factors.

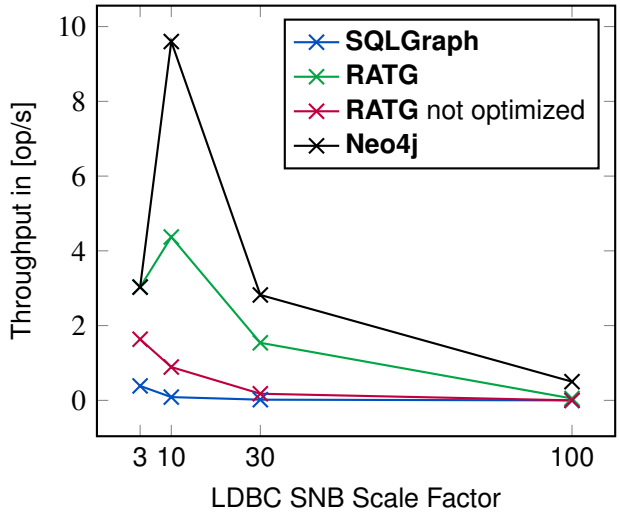



Figure 6.3.: Significantly improved throughput of the LDBC-SNB depending on the scale factor.

We want to emphasize that only **Neo4j** and **RATG** managed to successfully complete the workload at scale factor 100, while neither the unmodified **SQLGraph** approach, nor the not optimized version of **RATG** terminated.

While **Neo4j** also performs slightly better on most read-only queries, the majority of the throughput difference between **RATG** and **Neo4j** results from the difference of update queries. As we have stated before, our use case mainly relies on the efficiency of read-only queries and will only in extreme cases exceed the complexity of the data set with scale factor 3.

While we can see a difference in achieved throughput between **RATG** and **Neo4j**, most of the discrepancy is due to a few queries that perform worse on **RATG** than **Neo4j**. This also means that most of the queries perform comparatively well.

The LDBC-SNB (version 0.3.1) contains 14 *Complex Queries*, 7 *Short Queries* and 7 *Update Queries*. For the rest of this chapter, we will abbreviate these to *CQ1 - CQ14* for *Complex Queries*, *SQ1 - SQ7* for *Short Queries* and *UQ1 - UQ7* for *Update Queries*.

Let us now take a closer look at the read-only query performance of the different approaches, which was the focus and main assumption for our use case. In this chapter, we will mainly focus on the results we achieved using the data set of scale factor 10, because our results already show big enough differences for this data size. The general trend of the query performance between the different approaches does not change for bigger data sets, but the efficiency differences only get more extreme. Also, the results we obtained for scale factor 10 can be displayed in figures in such a way that the reader can obtain a general impression of the results, while for bigger data sets the scaling of the graphs can be a major challenge.

To review the results we obtained for all scale factors the reader is encouraged to consider Tables A.1 to A.6. For every run we used an operation count of 10,000 operations, following

several hundred warm-up queries. If a query type was not generated by the benchmark framework, we mark the times with −. The benchmark run that did not terminate (the SF100 run on **SQLGraph**) is marked with ↯.

### 6.3.2. Read Performance

As described before, we optimized **RATG** for read-only queries. Therefore, we expected our approach to perform significantly better for this type of queries. To confirm this we have evaluated our approach with the use of the LDBC-SNB and performed benchmark runs on data sets of the scale factors 1, 3, 10, 30 and 100.

**Short Query Results** Figure 6.4 depicts the results of our conducted tests for of all *Short Queries* of the LDBC-SNB (SF10), except *SQ2*. This type of query usually requires the system under test to evaluate a realtively low amount of vertices and edges to compute the result set. Typically, the neighbors of one entity for the data set is accessed [Erl+15].

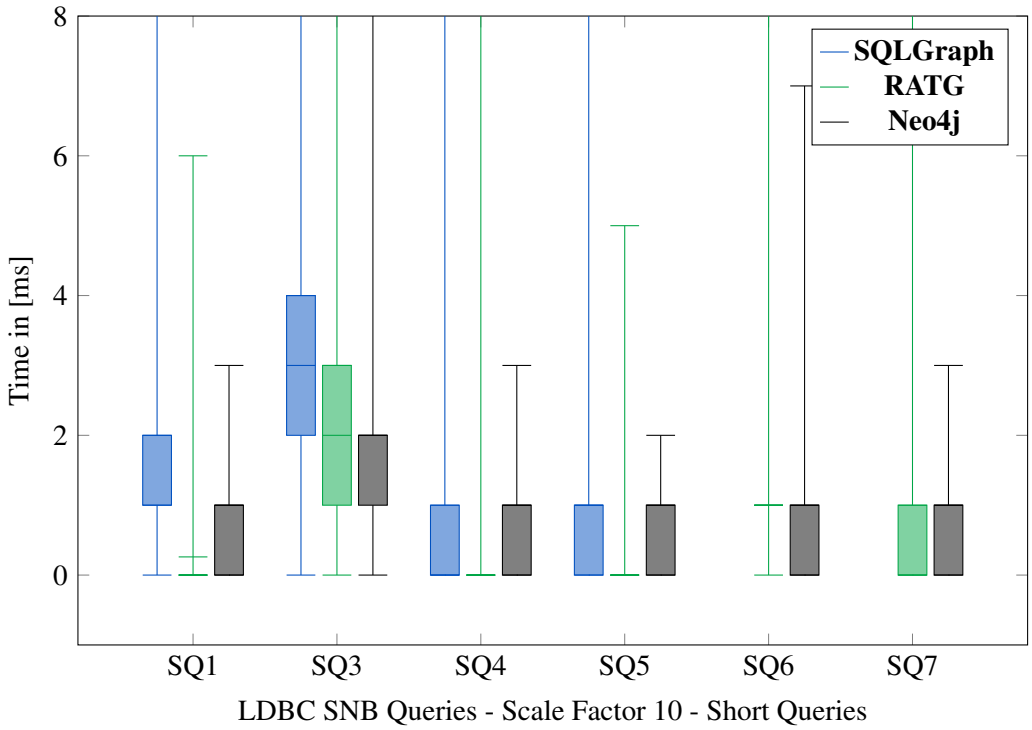



Figure 6.4.: Runtime of the Interactive Workload Short Queries - SF10.

We expected our approach to perform very well on these simpler queries, since the selectivity of the (sub-) queries is high, which is a strength of relational databases. Our findings show that **RATG** performs much better than **SQLGraph** in every *Short Query*.

*SQ1*, *SQ4* and *SQ5* select a single vertex and trace along a relation type that only has one target vertex. Especially in those queries we can observe the advantage of not requiring a join. Even compared to **Neo4j**, those queries perform exceptionally well. Note that *SQ4* and *SQ5* in the majority of executions are answered in less than a millisecond, which is the smallest unit we measured.

*SQ6* and *SQ7* search for a graph pattern that represents a path that contains three different vertices, while only *SQ7* contains a vertex with a high degree of edges of the queried type and in addition an optional edge that forms a cycle with the main path of the graph pattern. **SQLGraph** performed so significantly worse on *SQ6* and *SQ7* that the resulting execution times lie outside of Figure 6.4. As we expected, the more edge hops a query performs, we can observe a bigger difference in execution time. We can see that **RATG** still performs similarly to **Neo4j** while both outperform **SQLGraph**.

*SQ3* uses an undirected edge. Since **RATG** does not directly support undirected edges, this query results in a *SQL* query that uses both the incoming adjacency and outgoing adjacency tables and therefore is similar to a 2-edge-hop. In addition, *SQ3* selects a relation type with a high degree of neighboring vertices.

In comparison to the other short queries, *SQ2* produces a relatively huge intermediate result. While **RATG** outperforms **SQLGraph** by a factor over 30, **Neo4j** significantly outperforms both for this query. We did not include *SQ2* in Figure 6.4, because only the results for **Neo4j** would lie within the boundaries of the figure.

In conclusion one can see that our approach outperforms **SQLGraph** in every single *Short Query*, while it is still similarly efficient as the specialized graph database **Neo4j**. A result to keep in mind for the analysis of the *Complex Queries* is the fact that we found a major factor for the query performance of **RATG**:

- The more selective (sub-)queries are, the more efficient **RATG** is. If intermediate results become too big, our approach suffers in comparison to **Neo4j**.

With this in mind, let us now consider the *Complex Queries* provides by the LDBC-SNB.

**Complex Queries Results** To get an impression of the different runtimes of all *Complex Queries* consider Figure 6.8.

As we can see, *CQ2*, *CQ7* and *CQ8* (see Figure 6.5) take the least amount of time to compute. The common features these queries show, are the low amount of edge-hops that are required to compute their results. *CQ2*, *CQ7* and *CQ8* all only require 2 or 3 edge-hops using edges with relatively high selectivity (e.g. hasCreator which always only has one target vertex).

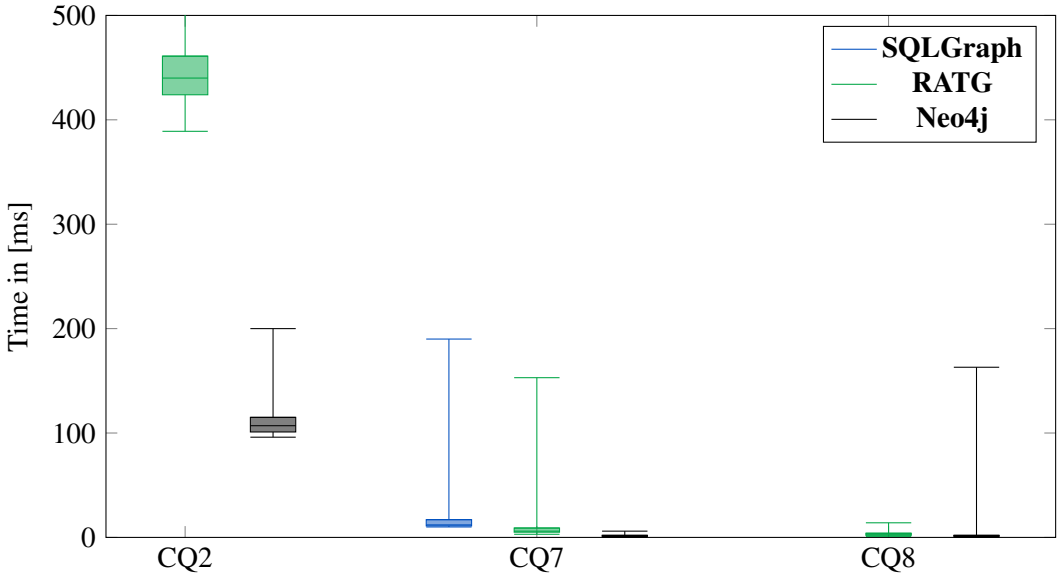



Figure 6.5.: Runtime of fastest Interactive Workload Complex Queries - Scale Factor 10.

Let us consider Figure 6.5 for the results of *CQ2*, *CQ7* and *CQ8*. All of those queries show the same efficiency: Our approach **RATG** is much faster than the original **SQLGraph** approach. So much so that our results for both, *CQ2* and *CQ8* lie outside of the figure. In all queries **Neo4j** performs slightly better than **RATG**.

Next, we want to take a look at *CQ6* and *CQ9*, which are depicted in Figure 6.6. The slowest group of queries consists of *CQ6* and *CQ9*, while *CQ6* is by far the hardest query to answer for all three systems that we tested.

*CQ6* performs a complex aggregation computation over Post vertices (of which a huge number of instances exist in the data set) that includes a friend-of-a-friend path and a comparison between two Tag vertices that are connected to each of the Post vertices. As we can see, **RATG** performs better and the response times are much more reliable than for **SQLGraph**, while both systems get significantly outperformed by **Neo4j** for this type of query.

*CQ9* looks for Message vertices created by a friend-of-a-friend. While [Cou20] states that the size of the result set of this query can vary hugely in size, our approach **RATG** as well as **Neo4j** show a very robust query evaluation time for this query. While **Neo4j** performs slightly better on *CQ6*, **SQLGraph** nearly takes twice as long to answer this query.

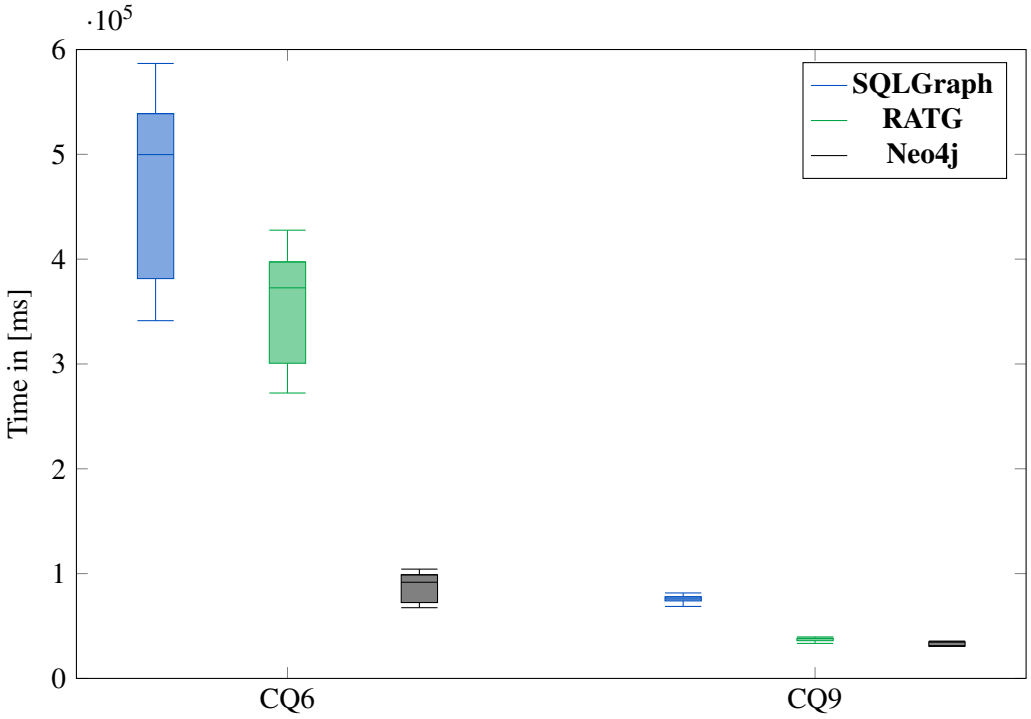



Figure 6.6.: Runtime of the slowest Interactive Workload Complex Queries - SF10.

By far the biggest group of queries is represented by *CQ1*, *CQ3*, *CQ4*, *CQ5*, *CQ10*, *CQ11*, *CQ12*, and *CQ13*, which all take a similar amount of time. Without an exception, this group of queries makes use of the knows edge type. In contrast to the other edge-types in the data set, the knows edge-type models an undirected relation. *CQ3*, *CQ4*, *CQ5*, and *CQ12* combine the use of the knows relation with some kind of aggregation of the number of Post or Comment vertices associated with those friends. The overall number of Post and Comment vertices in the data set is very high. Therefore, the selectivity of these queries is much lower than the selectivity of the fastest group of queries, but still higher than the selectivity of *CQ6* and *CQ9*. *CQ11* applies a similar pattern of searching for friends-of-a-friend, but looks for their work places. The complexity of the graph pattern of *CQ11* is relatively low and uses two edge hops that in most cases only have to consider a single edge (worksAt and locatedAt). *CQ10* and *CQ13* both make use of path patterns of variable length using the undirected knows edge-type. This leads to complex recursive queries that have to consider the edge list table in both directions. While one could suspect that this would make the queries slow, the relatively low amount of knows edges in the data still leads to a considerably fast response time for these two queries.

Overall, we can observe a general trend for this group of queries: For most cases our approach **RATG** performs better than **SQLGraph**, while **Neo4j** performs slightly better than **RATG**.

Queries *CQ3*, *CQ4* and *CQ5* display an interesting case: For these queries, **RATG** performs

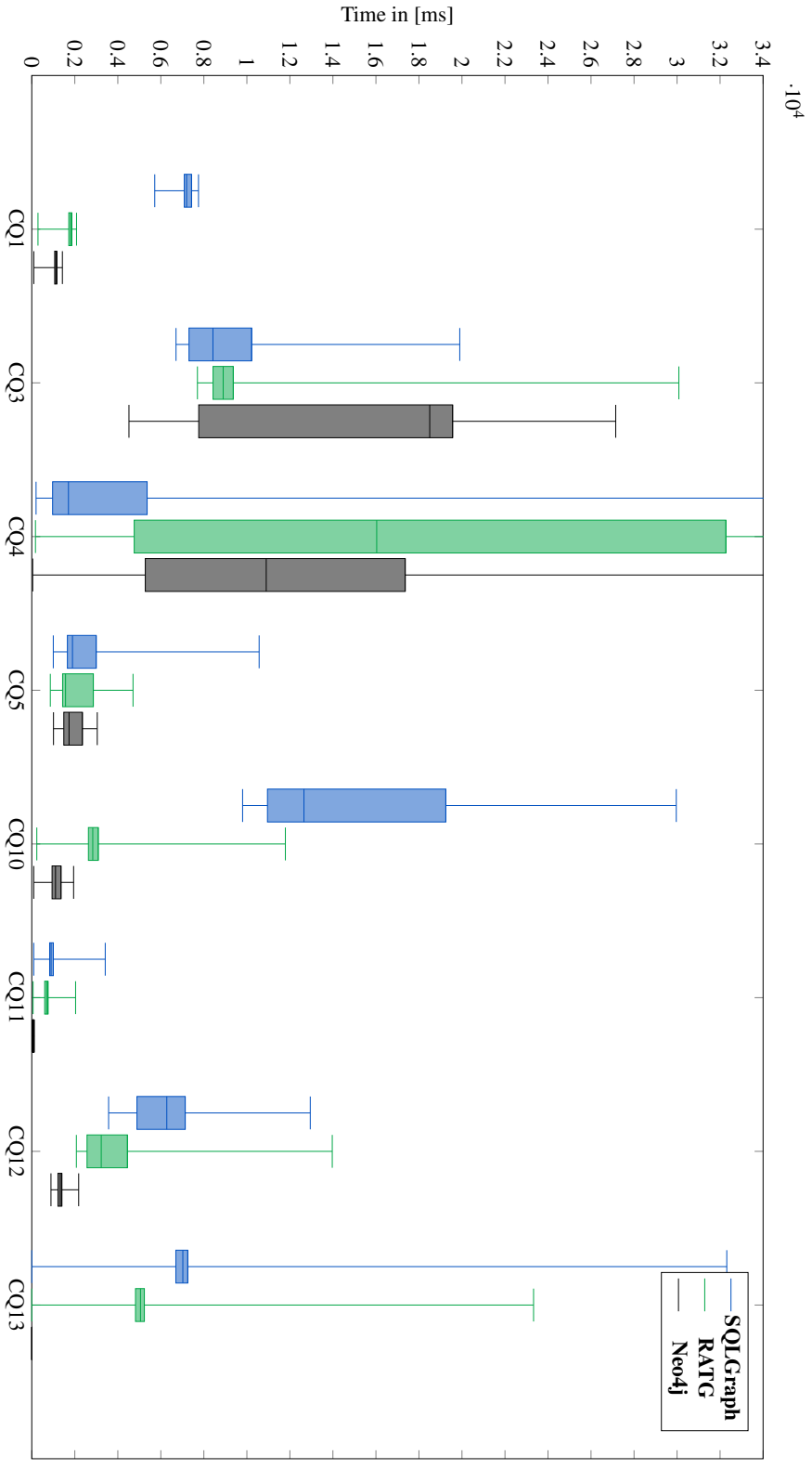



Figure 6.7.: Runtime of medium Interactive Workload - Complex Queries for SF10.

equally well as **Neo4j**, for *CQ3* **RATG** even performs better than **Neo4j**. These three queries are similar in their use of the hasCreator edge-type. Most Person vertices are associated with a huge amount of Comment vertices through the use of hasCreator edges. This can lead to possibly huge intermediary result sets. For *CQ3* and *CQ4* **SQLGraph** even performs better than **RATG**.

Looking at *CQ13* it is clearly visible that neither our approach **RATG**, nor **SQLGraph** can compete with a native graph database like **Neo4j** in regard to graph specific algorithms that have been implemented in the database itself: *CQ13* searches for the shortest path between two nodes of type Person. While **Neo4j** offers a predefined internal method to find this path, both **RATG** and **SQLGraph** have to rely on recursive *SQL* queries to find the result for this query.

To summarize, our findings confirm an increase in read performance of **RATG** compared to **SQLGraph** in almost all queries. We can observe this effect strengthening when the data set size increases. Considering complex queries, the difference in execution time becomes even more apparent, which confirms the findings in [Sun+15]. In their work, they stated that big intermediate join results (that are a point of focus for the set of complex queries) can be a bottleneck for their approach. Interestingly enough, for a few select queries like *CQ4*, this finding does not seem to take effect.

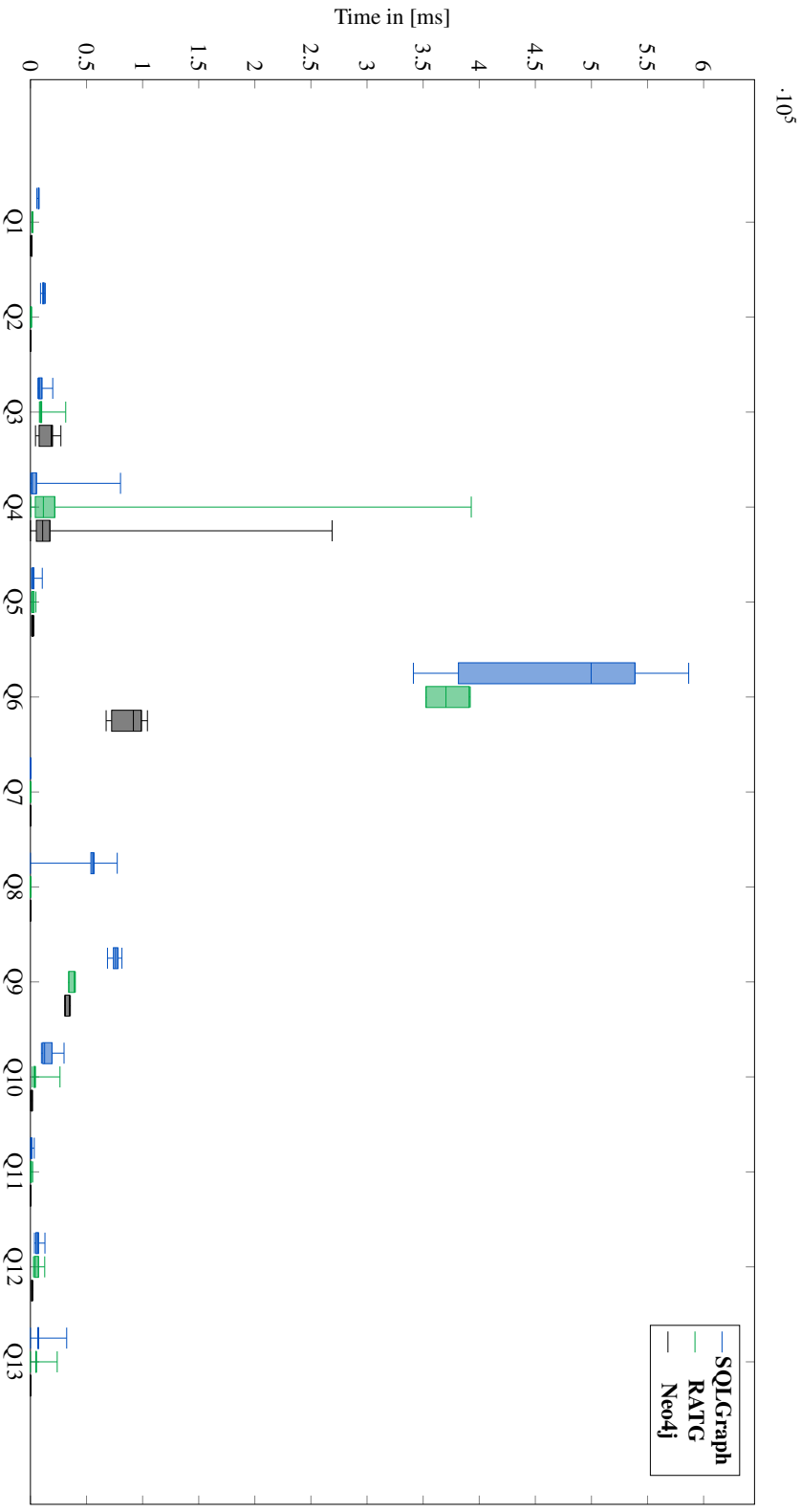



Figure 6.8.: Runtime of all Interactive Workload Complex Queries - Scale Factor 10.

### 6.3.3. Update Performance

As we have described above, we consciously chose to optimize **RATG** for read-query performance and therefore accepted a trade-off between read-query performance and update-query performance. We expected this trade-off to affect the time that is required to write updates to the graph. We will now present the results we obtained for update-queries provided by the LDBC-SNB. An overview of the results is depicted in Figure 6.9.

We implemented the update-queries using stored procedures (*PL/pgSQL*[2]). We decided to deviate from the usage of pure *SQL*, because the given update-queries can create several vertices and edges. In our schema the execution for these queries depends on several intermediary results (e.g. the internal VID of a Person node that is identified by its name). To avoid round trip times between the benchmark client and the database, we decided to use stored procedures. All the given update-queries are implemented using basic building blocks like *createVertex()* and *createEdge()*. Using specialized stored procedures for each query would most likely improve performance further.

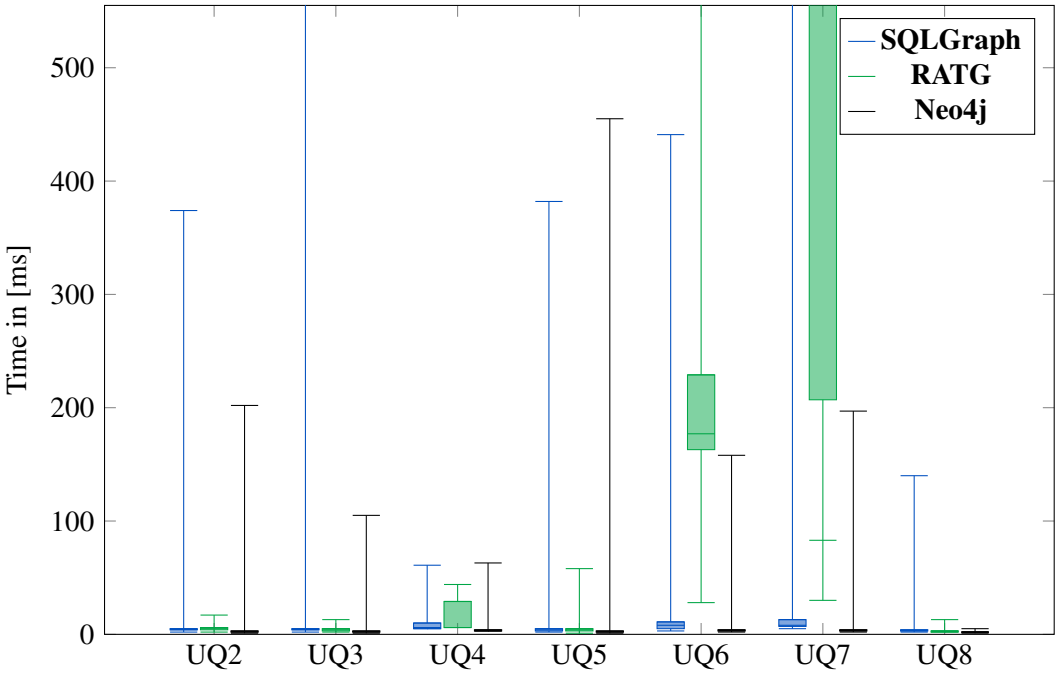



Figure 6.9.: Runtime of all Interactive Workload Update Queries - Scale Factor 10.

Let us first consider the five similarly fast executing queries depicted in Figure 6.10. The picture is not as clear as our targeted optimization goals would indicate.

UQ2, UQ3, UQ5 and UQ8 insert an edge between two already existing vertices. Therefore, for **RATG** the required time to insert the new edge depends on the amount of edges that have

[2]https://www.postgresql.org/docs/current/plpgsql.html

already been stored for the given vertices. **SQLGraph** only requires an insert-query to the secondary adjacency tables. In all three systems the execution time for this type of query only differs a few milliseconds. UQ4 creates a new vertex and inserts several new edges' between the new vertex and already existing vertices. Since the new edges target vertices do not have many connected edges, the insertion time remains very stable.

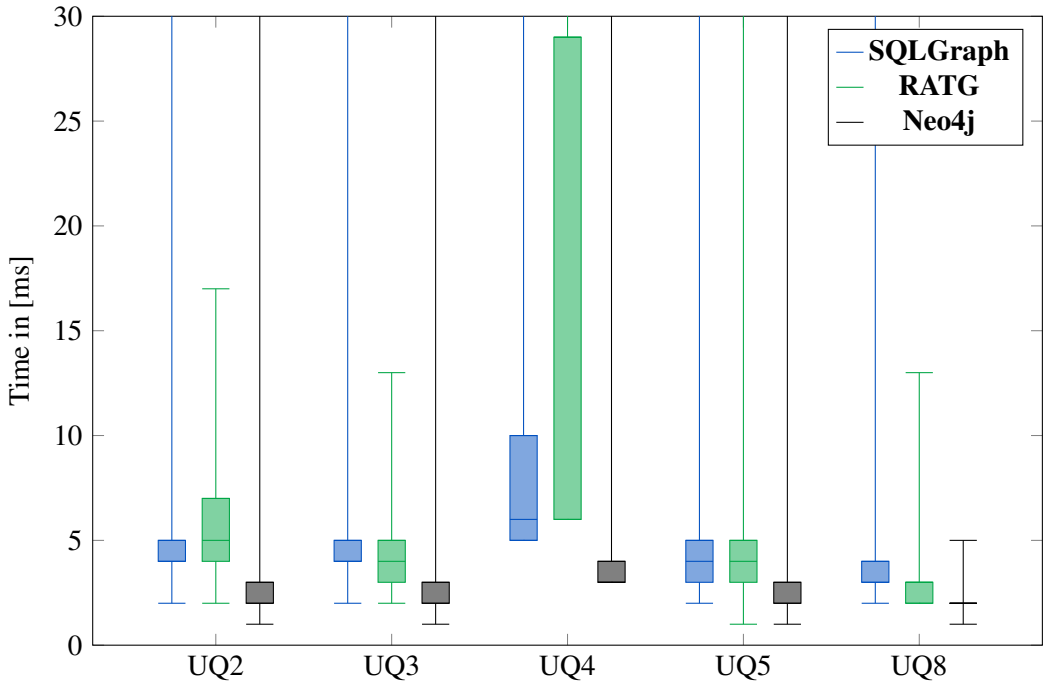



Figure 6.10.: Runtime of fast Update Queries - Scale Factor 10.

When we consider Figure 6.9, we can clearly see that the exeuction time of update-queries is dominated by two specific queries. **Update Query 6** and **Update Query 7**. A detailed view of the execution times of UQ6 and UQ7 is depicted in Figure 6.11.

We can clearly see that in the case of UQ6 and UQ7, **RATG** performs significantly worse than both **SQLGraph** and **Neo4j**. In contrast to the previously discussed update queries, these two queries create a new vertex and both introduce a number of new edges to the graph, as well as creating edges to and from nodes that already have a high number of edges attached. This means that for **RATG** a number of big arrays have to be altered, which explains the worse performance compared to **SQLGraph** and **Neo4j**.

In the case of **SQLGraph** the number of already existing edges of a node does not change the work required to insert a new tuple to the secondary adjacency tables, while **Neo4j** only has to alter the internal lists of edges [RWE15]. Overall, *UQ6* and *UQ7* show the behavior we initially expected from updates on **RATG**.

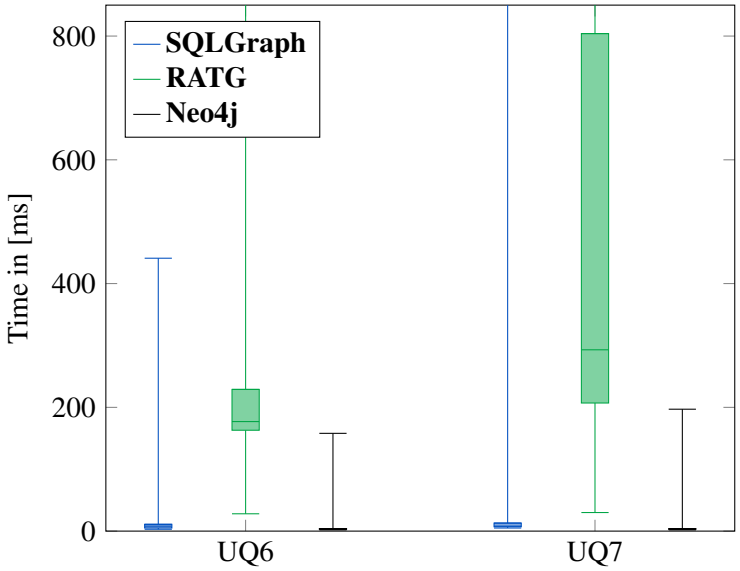



Figure 6.11.: Runtime of Slower Update Queries - Scale Factor 10.

In conclusion, we see that we can differentiate between two insert operation cases:

- For cases in which edges are added to vertices that do not yet have a big set of edges, our optimization does not negatively influence the performance of insert operations. Therefore, it performs well for most update queries of the LDBC-SNB.
- On the other hand, insert operations for vertices with a lot of edges are massively impacted. While a more efficient implementation of the applied stored procedures could reduce the difference in execution times, we do not think that the performance of the other approaches could be achieved for these types of operations.

In contrast to our assumption that our read-query optimization would slow down update performance, in many cases **RATG** performs as well or even better than **SQLGraph**, while none of the two approaches can perform better than **Neo4j** in regards to the efficiency of insert operations. This is not surprising, since **Neo4j** barely uses indexing (only for attributes) and does not have redundant storage.

## 6.4. Required Disk Space

Our approach reduces the number of tables in use. Therefore, we expected a reduction in the amount of disk space required to store the same data in comparison to **SQLGraph**. While the number of redundantly stored edges does not change, we can omit the storage of indices required to join the primary adjacency tables and secondary adjacency tables.

To confirm this, we imported different sizes of generated data sets provided by the *LDBC-SNB* data set generator. The generator creates data sets according to an input scale factor $(1, 3, 10, 30, 100, \ldots)$. We then sampled the required storage capacity using internal *PostgreSQL* functions.

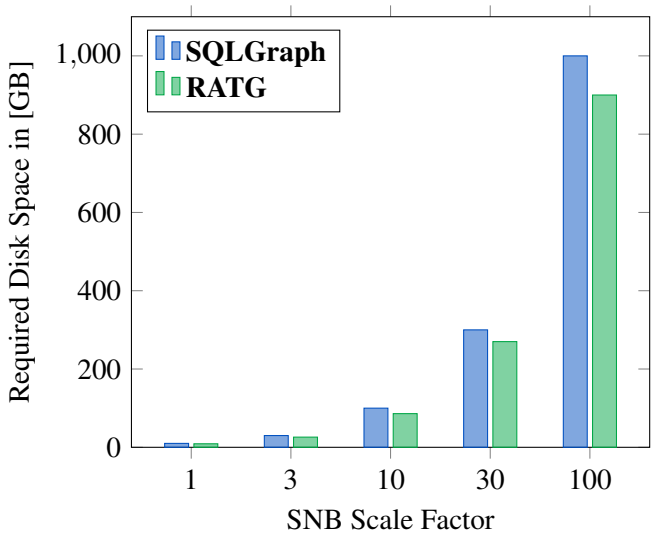



Figure 6.12.: Required storage capacity.

The numbers shown in Figure 6.12 represent the complete graph schema, including all indices and constraints. Our findings show that our optimization of the schema not only increases overall data retrieval efficiency, but also reduces the required storage space by over 10%.

As stated above, this is mainly due to the reduced overhead by reducing the amount of required tables and the associated index structures that are required to efficiently join the *outgoing primary table/incoming primary table* and *outgoing secondary table/incoming secondary table*.

## 6.5. LDBC - SNB: Conclusion

In this chapter we have described our methodology and results to evaluate the suitability of our optimized approach **RATG** and compared its performance to **SQLGraph** and the native graph database **Neo4j**.

Since our method reduces the number of joins that are required for most graph pattern queries, we expected our approach to perform much more efficiently than **SQLGraph**. Our evaluation confirms this assumption. At the same time, **RATG** performs very well compared to **Neo4j** on smaller data sets and reasonably well on huge data sets. The difference in throughput can mainly be traced back to single queries that perform significantly worse on our system.

The storage of edges in an array data structure leads us to the assumption that our approach should perform worse than **SQLGraph** on update queries. While this is true for the creation of edges at vertices that already have a high degree, we cannot confirm this for less used vertices. For a complete LDBC-SNB workload, our approach **RATG** still significantly outperforms **SQLGraph**.

Due to the reduction in the amount of required tables we also achieve a reduction in required index structures. This leads to a reduced disk space requirement of our approach.

Sun et al. [Sun+15] stated that one advantage over most native graph databases is the presence of sophisticated query optimizers. Our findings starkly contrast this statement: For nearly all of the LDBC-SNB queries, both the **PostgreSQL** optimizer, as well as the **OracleDB** optimizer (see [Sha21]), are unable to handle the complexity of these queries well. This can lead to intermediary results that are bigger than the available disk space and therefore queries that should be answered in a few seconds do not terminate. Therefore, manual optimization or a specifically implemented query optimization method for the translated *Cypher* queries is required.

The characteristics of the data originating from our use case fits the strength of our proposed solution: First, the building models we have investigated so far rarely exceeded a few hundred MB of data. The resulting data set therefore is smaller than the LDBC-SNB SF1 data set. Second, the degree of most vertices in the resulting data sets is not very high, but is more than a single edge for most relevant edge types. Third, most queries that we need to compute have the characteristics of LDBC-SNB *Short Queries* (for a detailed look at the queries for our use case see Section 10.2.2). For those **RATG** performed particularly well. As described in Section 3.2 we have optimized **RATG** for exactly this kind of data set.

Our evaluations show the suitability of our approach for the LDBC-SNB data set and therefore looked highly promising for our use case. For a detailed evaluation of a specialized version of our approach **RATG** on real world IFC data see Chapter 10.

# Part III.

# The Building Information Management Model Store

# 7. Use Case: Building Information Modeling

This research was triggered by the requirement to integrate three-dimensional building data into the *MonArch* system. We will first describe the *MonArch* system and then the type of data we want to integrate into the system, namely Building Information Modeling (BIM) data. Afterwards, we will describe the data model leveraged to achieve this goal: The Industry Foundation Classes (IFC).

## 7.1. The *MonArch* System

*MonArch* is an information system designed for documenting structures such as architectural objects, urban situations, and archaeological sites. A *MonArch* database consists of a digital model of the structure, i.e. a digital representation of the building, ensemble or site, together with a (potentially huge) body of information and digital documents. Any digital or digitized information or document can be attached to the digital model or specific parts of it. Thus *MonArch* provides a space-related organization of information, documents and artefacts [FS17].

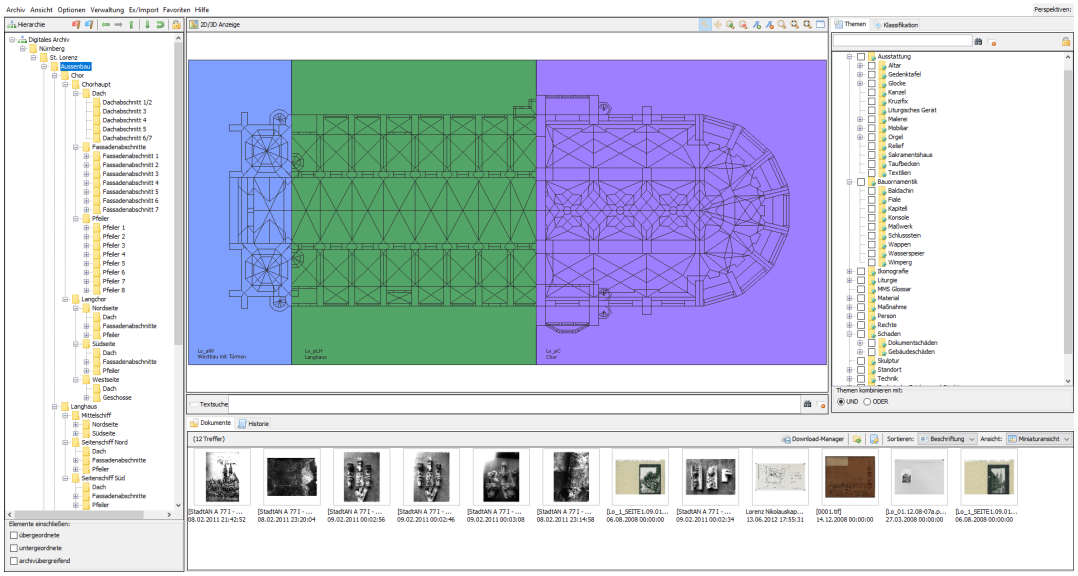


Figure 7.1.: The user interface of *MonArch* displaying the 2D navigational map.

So far, the digital representation of the model contained a two-dimensional building plan (also called navigation plan), a hierarchical structural representation of the building and a topical context. The hierarchical structure and the building plan are linked in such a way that both can be used to interchangeably navigate through the building. Each document can be assigned to multiple structural objects and topics. These can then be used to find the documents intuitively by selecting the part of the building and the topics the user is interested in. The user interface of the **MonArch** system is depicted in Figure 7.1. For a detailed description of the *MonArch* system and its data model we would like to refer the reader to Stenzer [Ste18].

One of the **goals of this thesis** was to extend the capabilities of the system and realize the **seamless integration** of **3D building models** that can then be used for navigation through the building and hence through the body of documents.

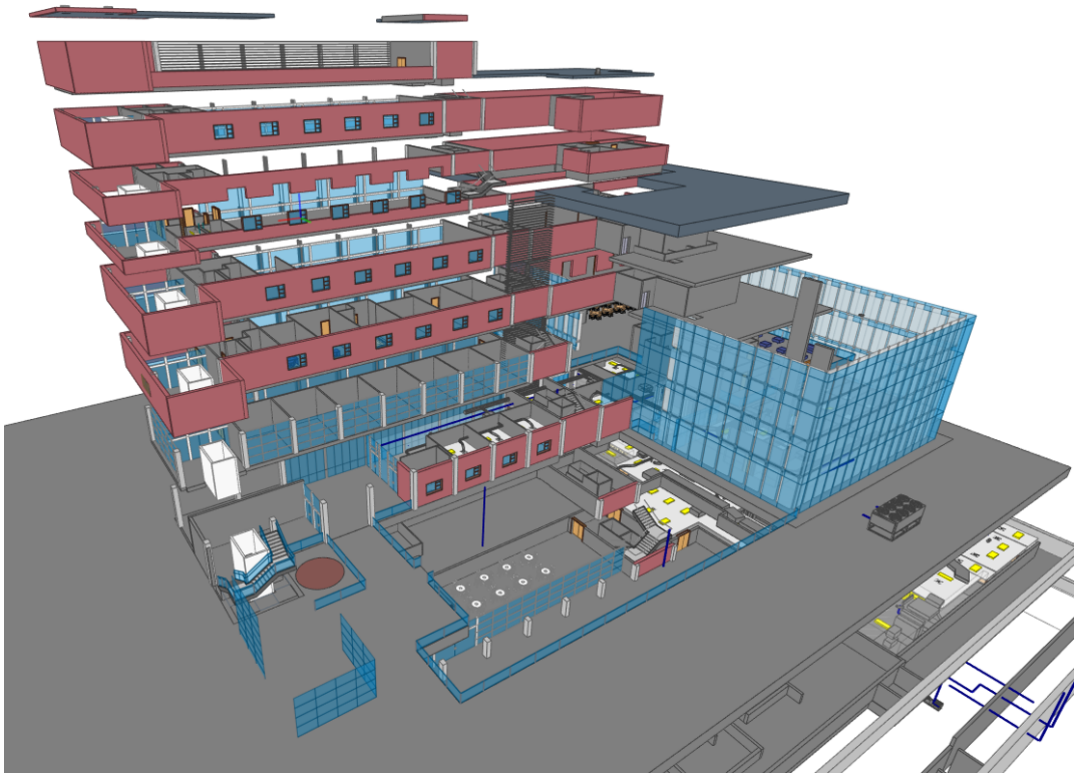


Figure 7.2.: An exploded view drawing of a 3D building model.

The current version of the data model of *MonArch* is implemented in the relational database management system **PostgreSQL**. To achieve the seamless integration we had to meet the following conditions:

- The building model data has to be stored in the relational database system.
- To achieve the linking of additional information with the building model, we have to be able to split the model into different segments. Optimally, these segments structure

the building hierarchically starting at the whole and stopping at specific components and parts. An example of such a segmentation can be seen in Figure 7.2 depicted as an exploded view drawing of the components.

- Each of the objects resulting from the aforementioned segmentation needs to be identifiable in order to be able to reference the segments and therefore be able to link additional information, e.g. documents or topics, to these.

To achieve the goal of integrating 3D building data into the *MonArch* system, we used models resulting from the Building Information Modeling approach. Therefore, we will now give a short overview of what BIM represents.

## 7.2. Building Information Modeling

Contrary to popular belief, BIM is not a software or a collection of different applications but rather a general approach to the continuous use of digital building models. In theory, these models should be used to manage the entire life-cycle of a building which starts at the early conceptual design phase, accompanies the construction phase, the phase of operation and ends with a potential demolition of the building.

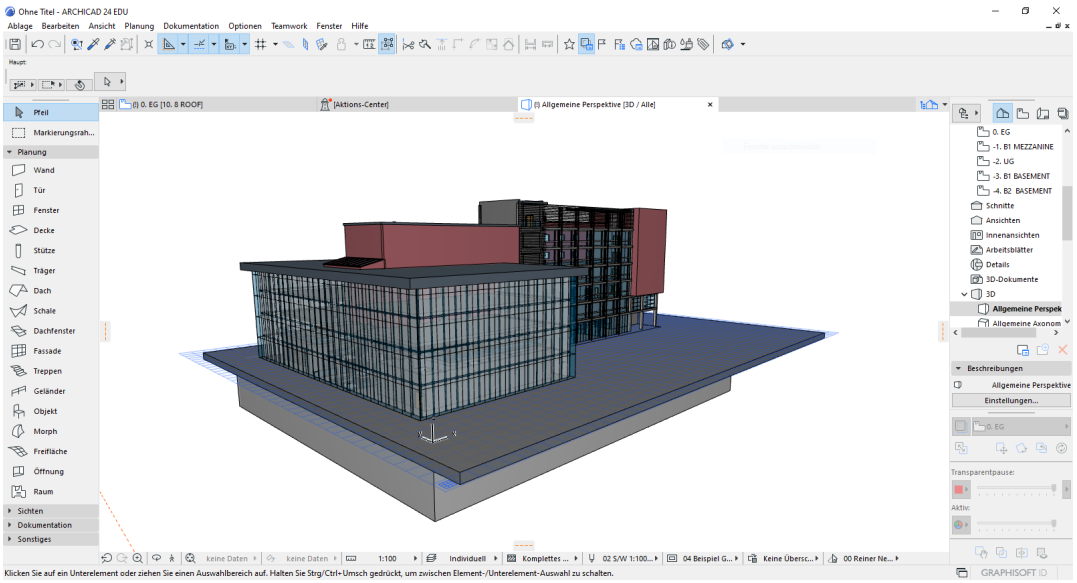



Figure 7.3.: A rendering of a *Conference Center* in *Archicad*.

In modern BIM, the central tool is a comprehensive digital representation of the building and includes great depth of information. This comprises extensive structural and semantic information about the building, as well as a three-dimensional representation of the facility [Bor+18].

These three-dimensional building models are created in specialized CAD software like Autodesk *Revit*[1], Graphisoft *Archicad*[2] by highly trained experts. A rendering of the BIM model of a *Conference Center* in *Archicad* is depicted in Figure 7.3.

In order to use a single central building model, the data exchange is necessary between different specialized software tools (possibly from different vendors). Because of this, some of these BIM software vendors have agreed upon a standardized open data exchange format and started to develop the Industry Foundation Classes (IFC).

## 7.3. The Industry Foundation Classes (IFC) Data Model

The Industry Foundation Classes (IFC) are a data model developed by **buildingSmart**[3]. It was developed to serve as an open Building Information Modeling (BIM) exchange format that can model a complete building. Therefore, it does not only include geometric data, but also information about the building components, their associated properties and relationships. It is registered with the **International Organization for Standardization (ISO)** as *ISO 16739* [ISO18]. IFC is an object-oriented and semantic data model that contains several hundred classes. These are organized in a complex hierarchy that includes numerous abstract layers [The11].

As described before, the IFC data model is a complex hierarchically structured data model. This makes it difficult for application developers to correctly use the data model, since the hierarchy contains a high number of abstract elements.

Within the scope of this thesis, the amount of classes and the complexity of the IFC data model is too large to discuss in detail. Instead, we will take a look at a few examples. To get a general impression of the number of different modules in which the IFC classes are organized consider Figure 7.4[4]. The approach proposed in this thesis generalizes from the concrete data instances and works on a property graph model of the data, which makes handling this number of different classes possible in the first place. We will describe this approach in the following sections.

**Remark 7.1**
For the remainder of this work, we will color the textual representation of abstract IFC classes in brown (e.g. *IfcProduct*), classes that can be instantiated in blue (e.g. *IfcWindow*), and objectified relationship classes in green (e.g *IfcRelConnectsElements*). □

[1] see https://www.autodesk.de/products/revit
[2] see https://graphisoft.com/de/archicad
[3] https://www.buildingsmart.org/
[4] Figure taken from https://standards.buildingsmart.org/IFC/DEV/IFC4_3/RC2/HTML/

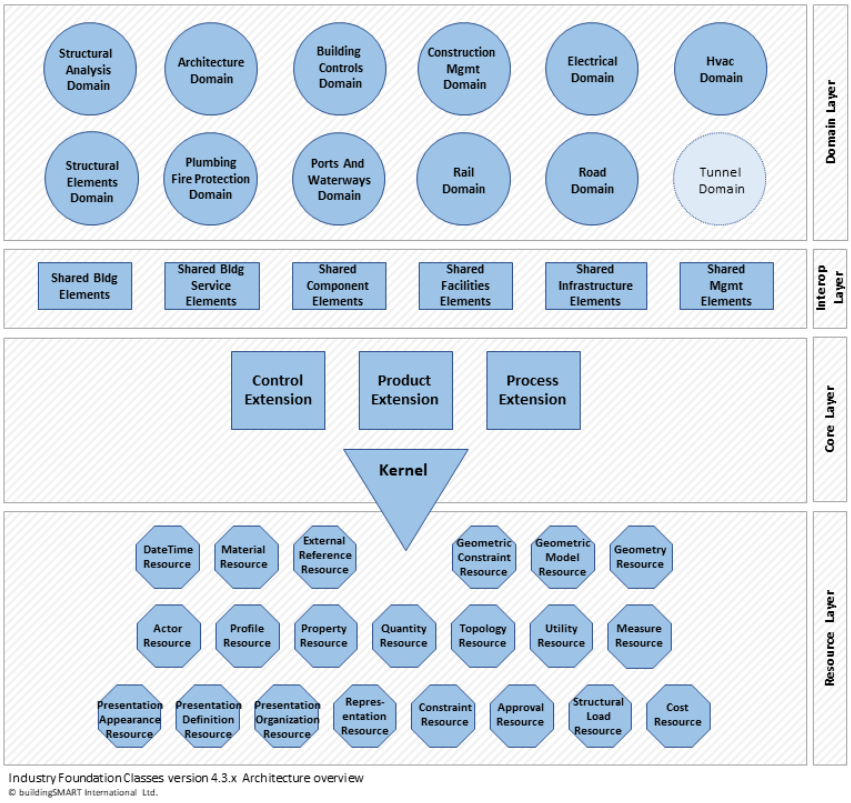



Figure 7.4.: IFC version 4.3.x Architecture overview.

### 7.3.1. The EXPRESS Data Modeling Language

Since the Industry Foundation Classes are defined using the *EXPRESS* data modeling language [SW94], we will give a very brief overview of the modeling language so that the reader of this work is able to understand our basic mapping of IFC data to a property graph model. The *EXPRESS* data modeling language is registered with the International Organization for Standardization (ISO) [ISO04].

The *EXPRESS* data modeling language is a data modeling language for use in engineering data exchange standards. It combines the entity-attribute-relationship and object modeling paradigms [BL01]. Objects of the real world are then described as strongly-typed entities. These entities then describe objects with common properties. Properties are modeled with attributes and constraints [Her94]. For this work we will not consider constraints defined in *EXPRESS*, since we assume that the data sets we receive are syntactically and semantically correct.

We can describe an entity using *EXPRESS* by using four types of attributes [Her94]:

**Simple data types** are attributes that are defined as values like numbers or strings.

**Collection data types** are also called aggregates. We can differentiate between *ordered* aggregation data types like lists or arrays, and *unordered* data types like sets.

**Named data types** can denote user-defined data type attributes (especially other entities) that are then referenced using an identifier. This is the only way to model a relationship to another entity using *EXPRESS*. Note that a collection data type can contain a collection of references.

**Inverse attributes** define that this entity is stored as a named data type attribute in another entity. Although, it is not required to define an inverse attribute in the target entity of a reference, this enables the restriction of the cardinality of the reverse direction of relationships. Also, if the *EXPRESS* data model is directly translated to an object-oriented class implementation, this allows for simple traversal of relations in the inverse direction.

An example of an *EXPRESS* definition is depicted in Listing 7.5: The *EXPRESS* definition describes the entity *IfcProduct*, which is an abstract representation of any object that relates to a geometric or spatial context [ISO18].

The ABSTRACT keyword defines this class as abstract and by using the SUPERTYPE OF construct, all subtypes of *IfcProduct* are listed (e.g. *IfcStructuralItem*). *IfcProduct* is a subtype of the (abstract) entity *IfcObject*. For *IfcProduct* itself, two attributes are defined:

1. **ObjectPlacement** is an OPTIONAL reference to an *IfcObjectPlacement*, which is used to define a placement for objects. This means that there can be a single reference or none;
2. and **Representation** is an OPTIONAL reference to an *IfcProductRepresentation*. The *IfcProductRepresentation* can, for example, be used to assign a geometric representation to a product.

As described before, the definition of an entity can also describe INVERSE relationships. This is a way to restrict the cardinality of *IfcRelAssignsToProduct* relations between entities. *IfcProduct* defines two INVERSE relations:

1. **ReferencedBy** is a SET (and therefore unordered collection) of references from *IfcRelAssignsToProduct*. This objectified relationship is used to handle the assignment (of subtypes) of *IfcObject* to (subtypes of) *IfcProduct* entities. It also defines that **ReferencedBy** is the *inverse* reference for the **RelatingProduct** attribute of the *IfcRelAssignsToProduct* entity.
2. **PositionedRelativeTo** is defined as a SET of references from *IfcRelPositions*. This objectified relationship is used to position a (subtype of an) *IfcProduct* relative to an *IfcPositioningElement*. The cardinality of the set is bound by a maximum of 1. Therefore, an *IfcProduct* entity can only be positioned relative to a maximum of one

*IfcPositioningElement*. It is also defined that **PositionRelativeTo** is the inverse attribute to the **RelatedProducts** attribute of the *IfcRelPositions* entity.

The WHERE clause can be used to define constraints for the entities. Since we will not use this in our work, we will not describe this further. The interested reader should consider [ISO04] for a more detailed description.

**The *STEP* File Format** IFC data is usually exchanged using the *STEP* file format, but an *XML* representation can also be used. The *STEP* file format is a text based human readable file format that is also registered with the International Organization for Standardization (ISO) as a group of standardized exchange formats [ISO21]. Lines are terminated by a ";" and every line of a *STEP* file describes a complete instance of an *EXPRESS* entity.

An example for an *IfcWallStandardCase* represented in *STEP* directly taken from one of our evaluation models is depicted in Listing 7.6: The line starts with the identifier *#28966* of the entity that will be described, followed by the entity type *IfcWallStandardCase*. Then the global identifier "3_YR89Qiz6UgyQsBcI$FJy" is defined, followed by a reference to the entity with identifier *#42*.

**Remark 7.2**
*STEP* files do not contain any schema information. Also, inverse attributes as described in Section 7.3.1 are not represented in the *STEP* file. Since we do not want to include inverse relations into the property graph representation anyway, this does not restrict us, and we also do not have to compute inverse relations at import time. □

```
ENTITY IfcProduct
  ABSTRACT SUPERTYPE OF (ONEOF(IfcStructuralItem, ...))
  SUBTYPE OF (IfcObject);
    ObjectPlacement : OPTIONAL IfcObjectPlacement;
    Representation  : OPTIONAL IfcProductRepresentation;

  INVERSE
    ReferencedBy : SET OF IfcRelAssignsToProduct
                                        FOR RelatingProduct;
    PositionedRelativeTo : SET [0:1] OF IfcRelPositions
                                        FOR RelatedProducts;

  WHERE
      PlacementForShapeRepresentation :
        (EXISTS(Representation) AND EXISTS(ObjectPlacement))
END_ENTITY;
```

Listing 7.5: The *EXPRESS* definition of the abstract class *IfcProduct* in *IFC4x2*.

```
#28966= IFCWALLSTANDARDCASE('3_YR89Qiz6UgyQsBcI$FJy',#42,
                  'Basic Wall:Reinforced Concrete"[...]);
```

Listing 7.6: The *STEP* file representaiton of an *IfcWallStandardCase* in *IFC4x2*.

### 7.3.2. General IFC Structure

As described before, the IFC contains hundreds of (abstract) classes and relationships that are objectified as classes. We will now take a quick look at an example that illustrates how building data is stored using IFC and should be sufficient to be able to understand our approach of storing IFC data in a relational database.

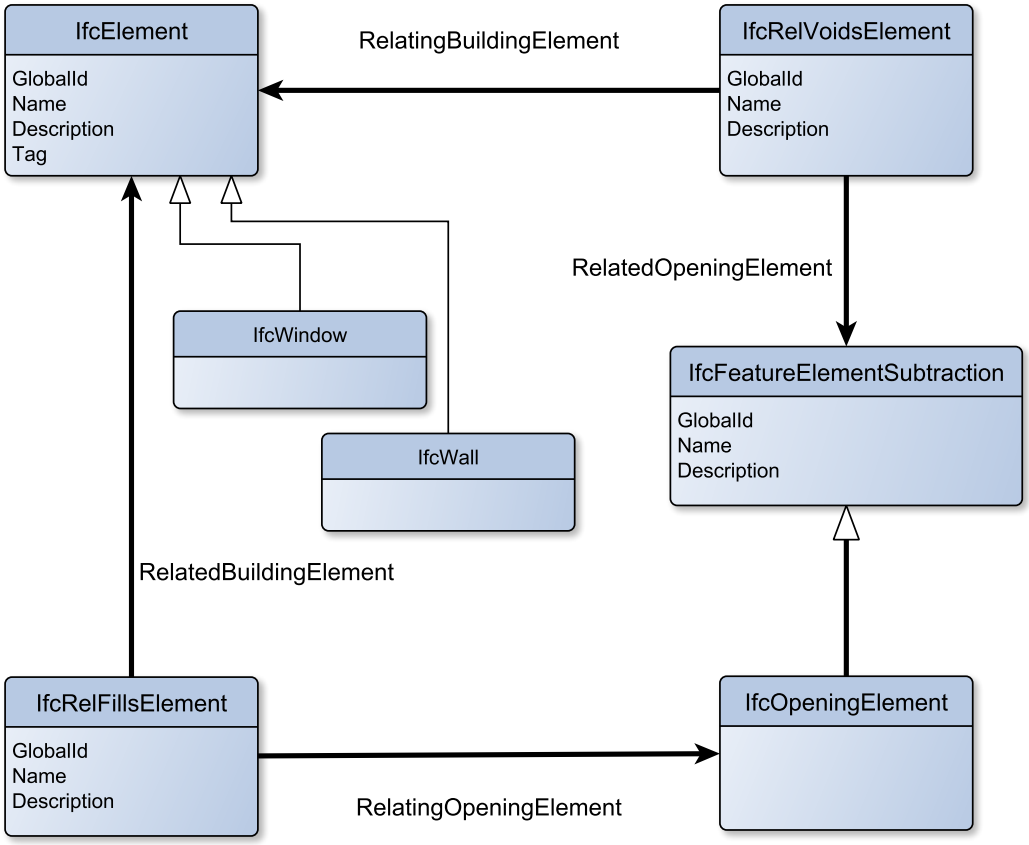



Figure 7.7.: Industry Foundation Classes (IFC) data model excerpt.

Consider the example excerpt of the IFC data model depicted in Figure 7.7. The figure shows the required classes to model a window that is placed inside a wall. Both classes, *IfcWall* and *IfcWindow* are subclasses of the *abstract* *IfcElement*, while the relations *IfcRelVoidsElement* and *IfcRelFillsElement* both are subclasses of the *abstract* relation *IfcRelation*. Note, that both classes and relations are implemented using classes known from object oriented languages. The actual relationships between those are then constructed using references.

In order to place the *IfcWindow* in the wall we need to create an opening in the wall. We can achieve this using *IfcOpeningElement*. We need to use *IfcOpeningElement* instead of its superclass *IfcFeatureElementSubtraction*, because we can only fill an *IfcOpeningElement*

with another *IfcElement* but the schema does not allow us to fill an *IfcFeatureElementSubtraction*. We can place the *IfcOpeningElement* in an *IfcElement* using an *IfcRelVoidsElement* relation that has references to both, the *IfcElement* and the *IfcOpeningElement*. Using the *IfcRelFillsElement* relation, we can then place the *IfcWall* in the *IfcOpeningElement*.

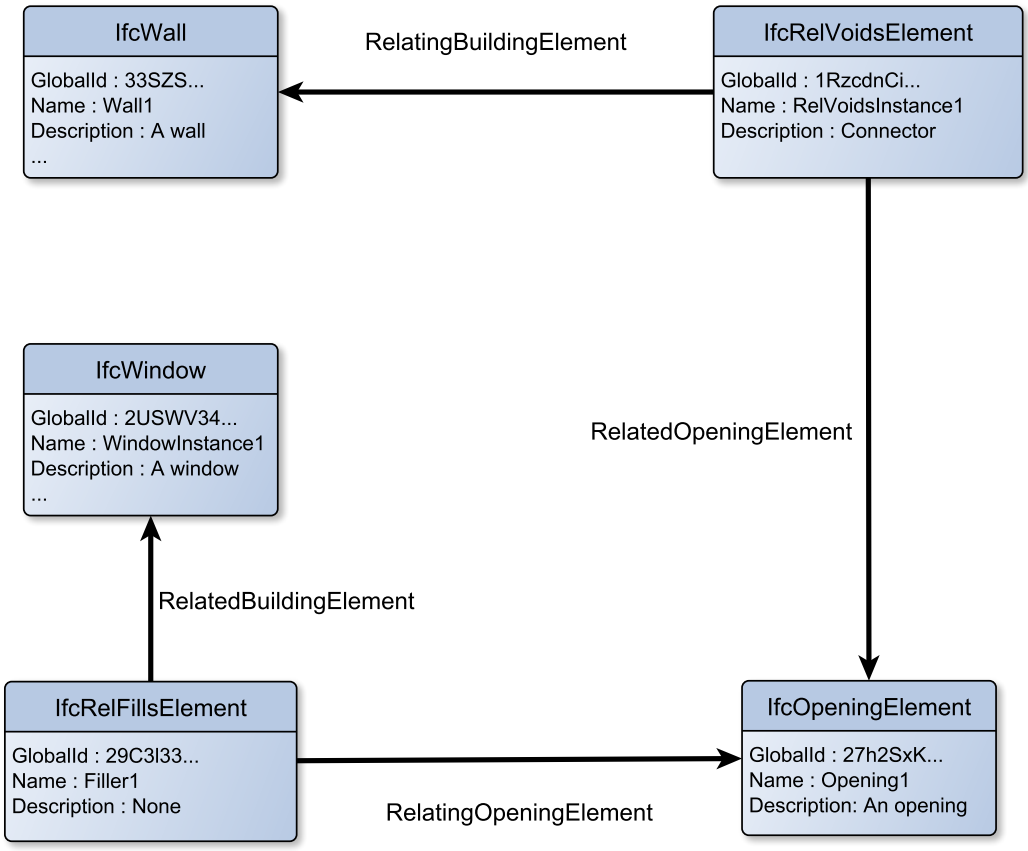



Figure 7.8.: Industry Foundation Classes (IFC) instance example placing a window in a wall.

An example of the instances that could be used to create the aforementioned example that places a window inside a wall is depicted in Figure 7.8.

**Remark 7.3**
We do not include inverse relationships in our examples and excerpts of the IFC data model, since we will not make use of inverse relationships in our approach. Because the property graph model allows for simple traversal of edges in both directions, we thereby reduce the number of edges required to map the data into our storage model while not losing any information. For the complete definition of the IFC entities please consider the official IFC documentation. □

# 8. Related Work

In this work we focus on the relational-based storage of IFC data. We do not consider closed vendor specific stores that are only available within a single software vendors software landscape.

## 8.1. IFC Model Stores

The IFC data model is a standardized (ISO 16739:2013 [ISO18]) model for the representation of a building and its life-cycle. It was developed with the purpose of enabling the exchange of data for the information produced during the design, construction and remaining life-cycle of buildings. Since the file-based data storage obviously has a number of drawbacks, there have been several attempts at storing the data in different types of database systems.

Among few others, the most prominent IFC model store is the open source software **BIMServer** [Bee+10b; Bee+10a]. It uses a NoSQL approach to store IFC building data. The general approach they apply is to store every entry of the IFC data in a key-value-based database (Berkeley DB), using the internal object id (provided by the *EXPRESS STEP* file) as the key. They offer functionalities like model revisions and merging. An important feature that distinguishes the **BIMServer** from other approaches is the possibility to execute queries on the building models. Nevertheless, these queries either have to be written in a Java-based query framework created using the eclipse modeling framework, which requires **BIMServer** implementation specific knowledge, or the use of *BimQL* [MB13]. While the Java-based query framework has been discontinued, the use of *BimQL* is hardly documented for the system. The result of each query is exactly *one IFC file*. Therefore, no simple values can be the result of a query. This also implies that only a single revision of a single building model can be queried at a time. To the best of our knowledge, this system achieves the best performance (in the sense of import performance and query performance) compared to other approaches [Lee+14], but at the same time is very hardware resource demanding. In addition, **BimServer** is based on *Java 9* and an upgrade to the most current long term support version *Java 11* or newer versions is not foreseeable for the near future, since the single original developer is not taking part in the development anymore. Due to the aforementioned restrictions regarding queries and hardware, this approach is not viable to our use case.

Lee et al. present an Object-relational Database Management System (ORDBMS) approach to store IFC data [Lee+14]. They use the *BUCKY* benchmark [Car+97] to show the validity of their approach, which is a query-based benchmark designed for comparing the performance

of ORDBMS and RDBs, but not designed for IFC use cases [Maa09]. The query performance presented in [Lee+14] is overall not satisfactory and the ORDBMS approach does not seem viable for an interactive information system.

Solihin et al. [Sol+17] present another approach to store IFC data in the RDBMS **OracleDB**. They automatically transform the data into an established star schema model (see e.g. [Ada10]), which is well-defined within the data warehouse domain. The goal of this approach is to allow flexible and efficient queries. Therefore, the data is transformed with the goal of analysis, not considering later retrieval of the original data. The linking of additional data to the IFC data is not considered, either. In addition, the modification of the data is generally not possible. In addition, in [SEL17] integrates different geometric representations using GIS functionality with the aforementioned data-warehouse storage concept.

## 8.2. Querying IFC Data

To the best of our knowledge, no open system exists that enables the user to perform arbitrary queries on given IFC data. With the development of [MB13], the authors started an effort to create a domain specific, open query language for IFC data. This work has not been finished and its only reference implementation can be found in [Bee+10b]. Since it is designed for IFC specifically, the user needs to have detailed knowledge about the structure of the IFC data model.

General query mechanisms for *EXPRESS* data exist in the *Express Query Language (EQL)* proposed by Huang et al. [KHJ98]. It is designed as a generic query mechanism for *STEP* files. The language has a complex grammar and syntax and therefore does not offer the same advantages as our approach of intuitively writing queries in *Cypher*.

The *Partial Model Query Language (PMQL)* [Ada03] is a query language that provides selection, update, and deletion functionality that supports recursive expressions and has been a major influence for the development of *BimQL* [MB13]. Because of its domain specific nature, a potential user must have detailed domain-specific knowledge to be able to write queries in *PMQL*.

## 8.3. IFC as a Graph

Since existing approaches to query IFC data required predefined queries or schema-based filters and therefore a profound knowledge about the IFC data model, Tauscher et.al [TBS16] approach the usage of graph theory to extract knowledge from IFC building models. They propose an approach to extract a graph representation from the IFC data in order to perform queries, using for example Dijkstra's the shortest path algorithms. Their approach relies upon filtering out unnecessary relationships (that could also produce wrong results) at graph generation time. While this enables them to easily find e.g. the *IfcMaterial* related to an

*IfcDoors*, much of the information contained in the IFC model is not present in the graph. In contrast to our approach, the graph data structure is also not used as the main data model to store IFC data.

Ismail et al. [INS17; ISS18] transform the IFC data into the property graph model and use **Neo4j** as its database backend. In their work, they extract a subset of the IFC data, building a subgraph of the graph that would be stored in our IFC store. Additional information or geometric information is not included in the graph representation. In contrast to our approach, they do not directly store the IFC data in the property graph format and do not give a general mapping approach to transform the data. Also, only a subset of the data is accessible through the graph query interface.

# 9. Storing IFC in the Relational Database

We will now describe our approach to storing Industry Foundation Classes (IFC) data in a Relational Database Management System (RDBMS). Since our work does not focus on the creation or validation of IFC data, we assume that the following properties hold:

1. The IFC data that is presented is syntactically, geometrically, as well as semantically correct. When using any parser library to read IFC data, it is the parsers task to ensure the syntactical correctness of the IFC *STEP* file. The validation of IFC data in a semantical and geometrical manner is still a subject of ongoing research (see e.g. [YL21]) and out of the scope of this thesis.

2. As previously mentioned in Section 7.3.1, the *EXPRESS* modeling language offers functionality to describe constraints and other value dependencies for *EXPRESS* entities. We do not consider those for our data management and assume that those criteria are met at import time.

After having stated our assumptions for the given IFC data and introduced the data modeling language and file format, we can now describe how we map *EXPRESS STEP* data into the property graph model.

## 9.1. Mapping IFC into the Property Graph Model

As previously described in Section 7.3.1, the *EXPRESS* model language describes entities with four types of attributes. We therefore perform the following mapping:

We map each entity on its own vertex type. This means that we will have a vertex type of every class of IFC that can be instantiated. Then we have to map the four types of attributes that can be defined in *EXPRESS*:

**Simple data types** We convert these attributes associated to an entity instance to **properties of the vertex** that represents the *EXPRESS* entity and use the name of the *EXPRESS* entity attribute as property key.

**Named data types** Since these attributes represent relations to other *EXPRESS* entites, we map these data types to relations. The source of the relation is the current *EXPRESS* entity instance and the target is the referenced *EXPRESS* entity instance. We then assign the name of the attribute to the relation as its label.

**Collection data types** Since we map **simple data types** and **named data types** in different ways, we need to differentiate between two cases:

1. A collection of **named data types**: We do not differentiate between ordered and unordered collections for **named data types**. We store both as an array in the property set of the vertex representing the *EXPRESS* entity instance and assign the name of the attribute as property key.
2. A collection of **named data types**: We create a relation for each of the references included in the collection the same way as we created a relation for a single **named data type** attribute. If the collection is an ordered collection, we add a property to the relation that stores the position of the current reference in the collection.

**Inverse attributes** We do not directly represent inverse attributes in the property graph representation of the IFC *EXPRESS* data. Since each **inverse attribute** is the inverse of the attribute in the referenced *EXPRESS* entity instance, the other direction is already included in the property graph. Because we can traverse edges in the inverse direction, we do not need to explicitly store these.

Let us now consider an example to illustrate the mapping.

**Example 9.1**
Consider the previous IFC instance example again that describes an *IfcWindow* in an *IfcWall* depicted in Figure 9.2: The example contains 5 *EXPRESS* entity instances: An *IfcWall*

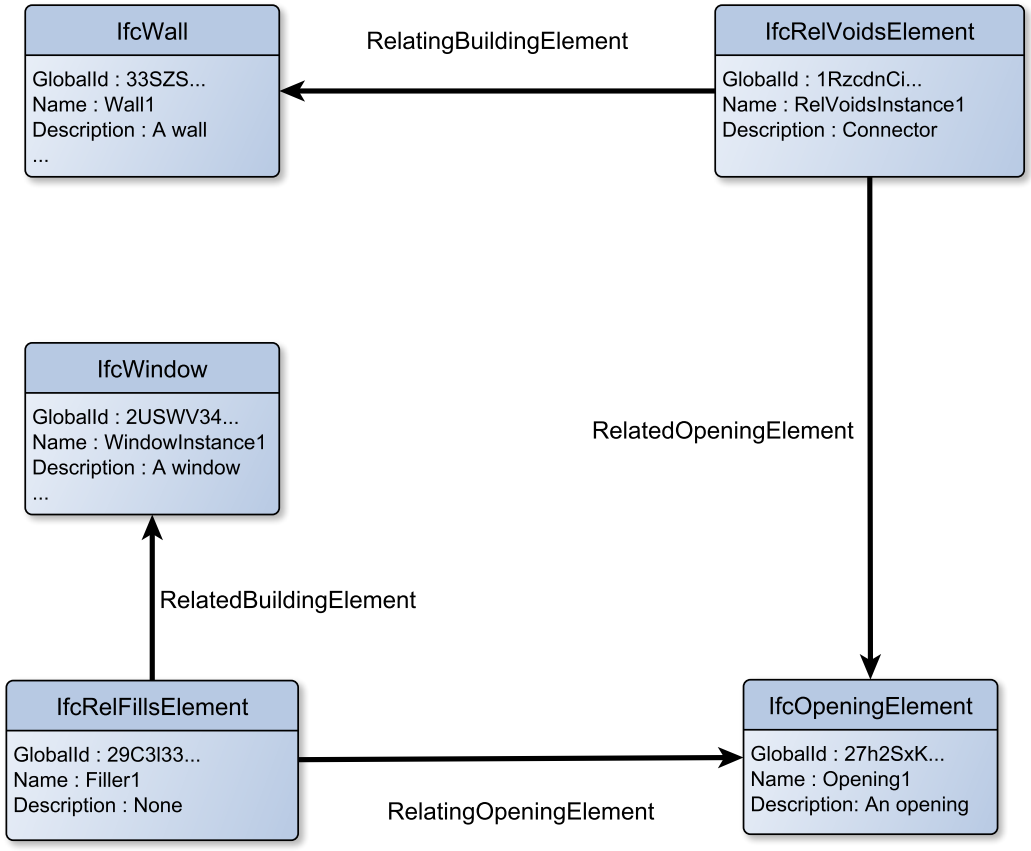



Figure 9.2.: Industry Foundation Classes (IFC) instance example placing a window in a wall.

instance that is referenced from an *IfcRelVoidsElement* instance. That instance references an *IfcOpeningElement*, which in turn is referenced by a *IfcRelFillsElement* that finally references the *IfcWindow* in the wall.

First, our mapping produces the vertices depicted in Figure 9.3. The intermediate result contains 5 vertices, each of which represents one of the *EXPRESS* entity instances of the data sample. **Simple data type** attributes and aggregation type attributes of **simple data types** have already been converted into property-value pairs.

For example, the *EXPRESS* entity instance of the IFC class *IfcWall* has been converted to a vertex with the automatically generated $VID = 1$ and the instance of *IfcRelVoidsElement* to the vertex with $VID = 2$. The *EXPRESS* entity type *IfcWall* has been converted into the key-value pair *type : IfcWall*. The **simple data type** attribute "GlobalId" has been converted into the key-value pair *GlobalId : 33SZS...*.

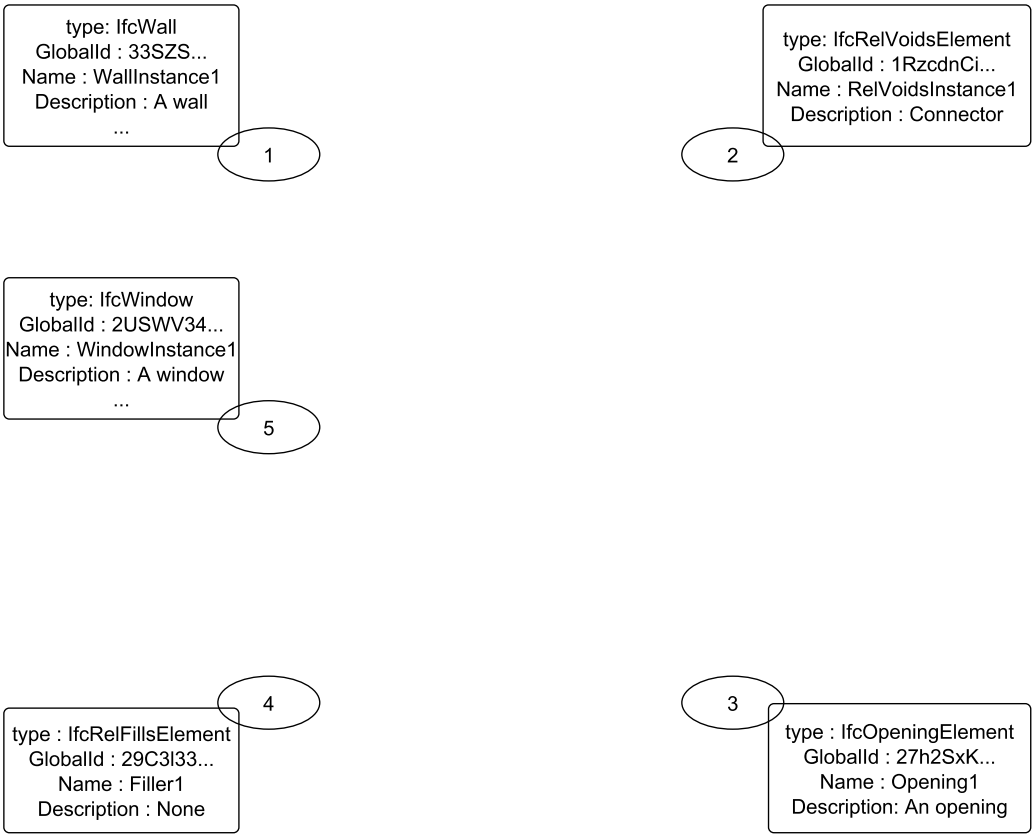



Figure 9.3.: Vertices resulting from the IFC instance depicted in Figure 9.2.

Second, we include **named data type** attributes and aggregation type attributes of **named data types**: The *RelatingBuildingElement* **named data type** attribute that describes the reference from the *IfcRelVoidsElement* instance to the *IfcWall* instance is converted to an edge from vertex $VID = 2$ to $VID = 1$, which is given the label RelatingBuildingElement. □

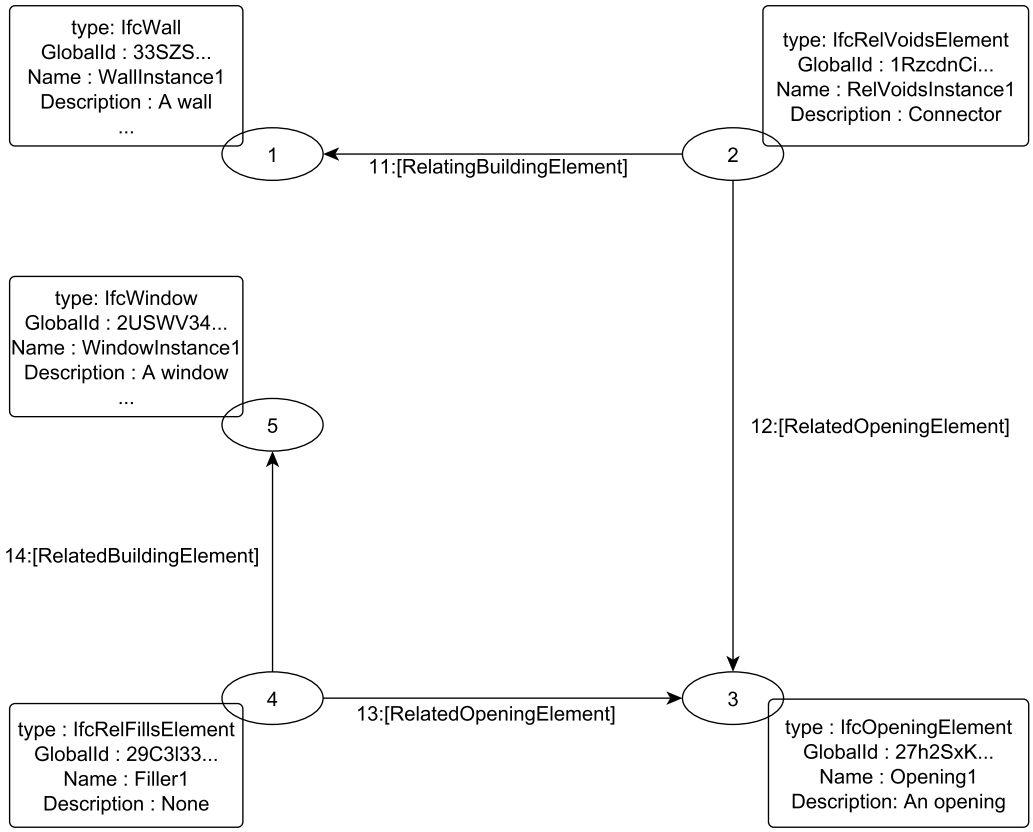



Figure 9.4.: Example property graph resulting from the IFC instance depicted in Figure 9.2.

**Example 9.5**
Let us additionally consider the small excerpt from the IFC data model depicted in Figure 9.6 to illustrate the mapping of **collection data type** attributes: This example shows how *IfcPolyloop* containing *IfcCartesianPoint* are described in IFC. The *IfcPolyLoop* contains a **collection data type** attribute of references to *IfcCartesianPoint* instances. Since the order of the points in a polyloop is essential, the collection is ordered. The *IfcCartesianPoint* contains

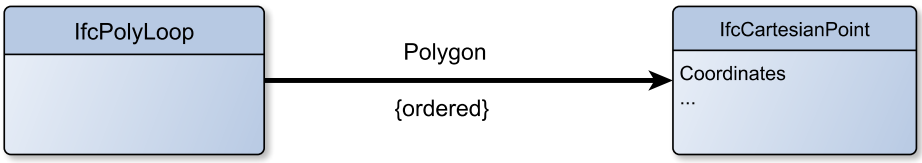



Figure 9.6.: Example of ordered references in IFC.

the **collection data type** attribute *Coordinates*, which contains a list of coordinate values that are **simple data type** values.

The mapping of **simple data type** attributes and **named data type** attributes works the same way as shown in Example 9.1. Since the *Polygon* attribute is a **collection data type** attribute that is also ordered, our mapping to the propery graph model results in the graph depicted in Figure 9.7. Several edges of the type Polygon originate from the IfcPolyloop vertex that represents the original *EXPRESS* entity instance. Those target the different *EXPRESS* entity instances referenced in the ordered collection. Therefore, we need to preserve the original order by introducing properties and specifying the original *position* of the target in the collection.

Additionally, the *Coordinates* attribute of the *IfcCartesianPoint EXPRESS* entity instance is an ordered collection of a **simple data type**. Therefore, we store these values as an array for the key *Coordinates* of our IfcCartesianPoint property graph vertex. □

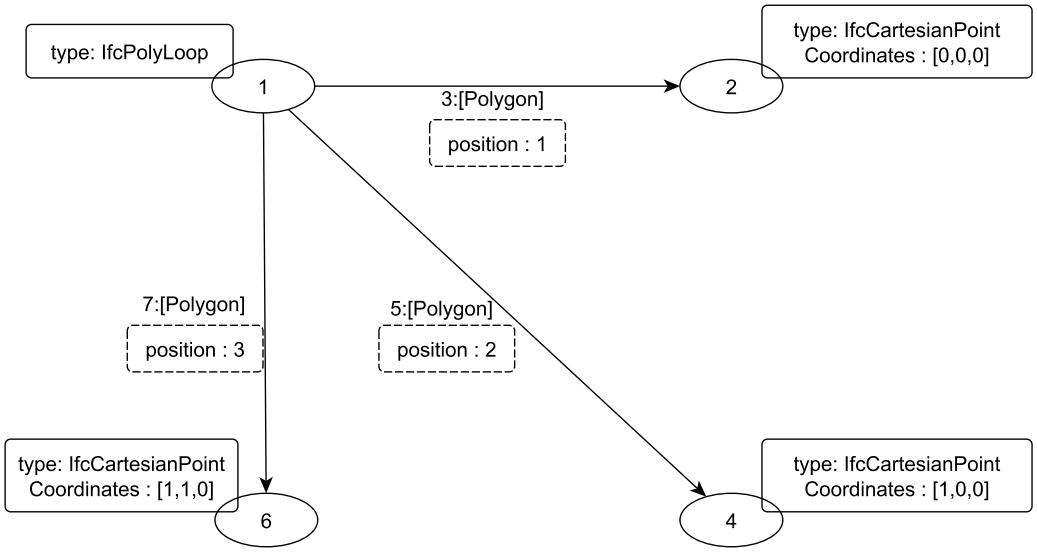



Figure 9.7.: Example property graph resulting from the IFC instance depicted in Figure 9.6.

## 9.2. Importing IFC Data

As we have described earlier in Chapter 4, the general concept to store data in our schema (depicted in Figure 9.8) consists of two processes:

1. We need to generate mapping functions in order to decide which type of relation is stored in which column triple. This is done in advance and is only necessary once per data model.
2. The actual data import then parses the IFC data, transforms it into a graph and writes the data to the database. For writing the data to the database, we apply the previously computed hash functions.

To compute the hash functions, we can use exactly the same approach previously that has been previously described in Section 4.1.1. Therefore, we only need to take a closer look at importing the actual data into the system.

As described in Section 4.2, our approach to efficiently import graph data into our schema is layered. This implies that we only have to provide the suitable parsing and the preprocessing layers to import the IFC data into our graph schema.

We can use existing parsing libraries for the IFC data to implement the parsing layer (e.g. the parsing component of BimServer [Bee+10b]). Therefore, for the IFC scenario we only need to create the preprocessing layer.

In our use case the preprocessing layer handles the transformation of the IFC data into the property graph model, using the mapping described in Section 9.1. Contrary to the import of LDBC-SNB data (as described in Example 4.17) we do not need to automatically generate identifiers for the vertices of our graph, because each *EXPRESS* entity described in a *STEP* file has its own unique (within this single model) id. Hence, we can use this identifier as our VID in the graph. Since we assumed that the input data is correct (see Chapter 9), we can furthermore assume that referential integrity of the references from an *EXPRESS* entity to another *EXPRESS* entity is not violated. Therefore, we can directly create edges as soon as we encounter them in the data, since we already know the VID of the target vertex in the graph and also know that this vertex exists.

For the sake of a clear description of the algorithm, we use a *list* of *EXPRESS* entities as input. In the real world application we use a stream of entities so that we do not have to keep the whole building model in main memory. A pseudo code description of our preprocessing layer is depicted in Algorithm 9.9.

Because we do not have to assign new identifiers to the *EXPRESS* entities in order to convert them to a vertex, we are able to import the IFC data in a single pass over the list of entities. Our internal representation builder only supports the creation of finished vertices, since we want to be able to stream the internal representation to the next layer (or for intermediate storage to the hard drive). Therefore, we gather all attributes in a set before we create the actual vertex.

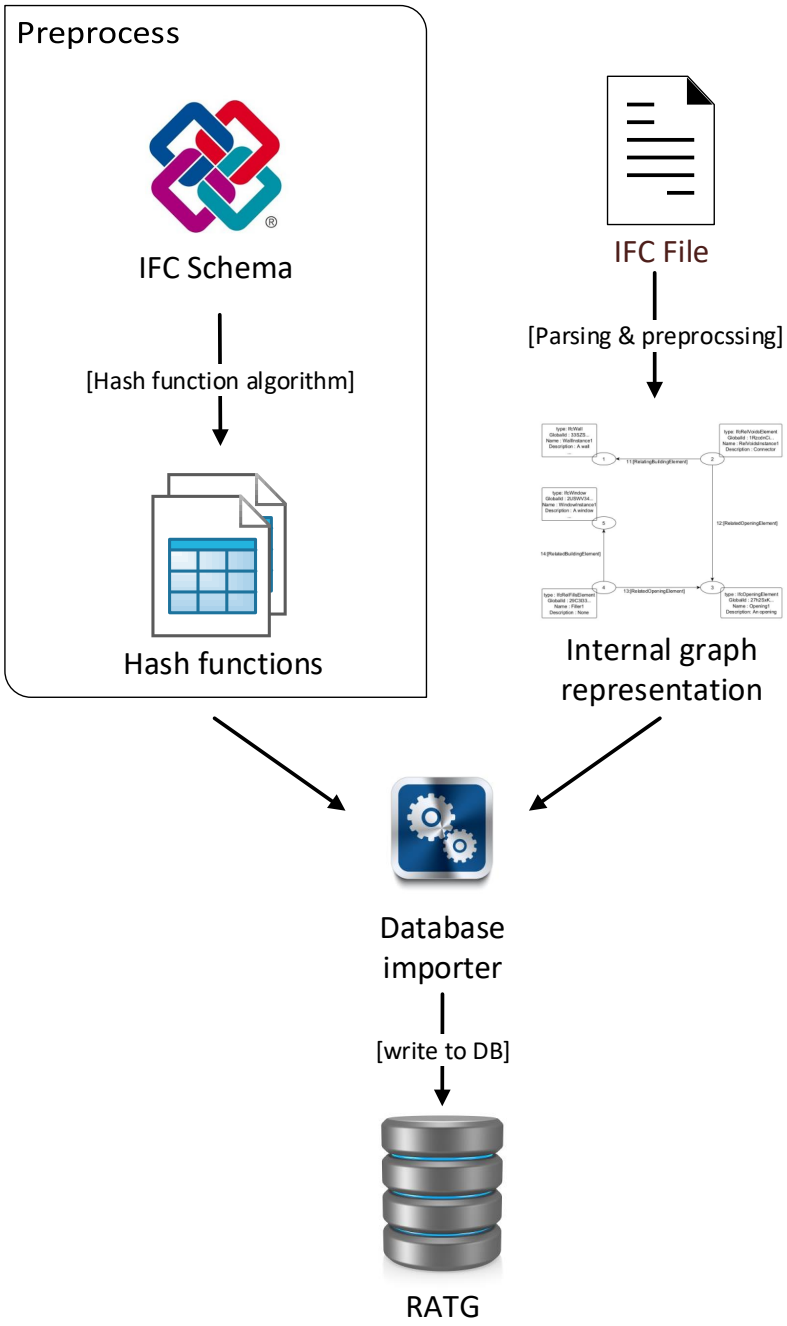



Figure 9.8.: Import concept for IFC data applying the concept presented in Section 4.2.

For each entity we process each attribute entry as follows: We check if it is a **simple data type** attribute. If this is the case, we store this attribute as a simple property in the property

**Algorithm 9.9:** The preprocessing layer of the IFC data import.

```
Input: The list of EXPRESS entities EX and a builder for the internal representation IRB.
/* Iterate over all EXPRESS entities.                                    */
foreach (e ∈ EX) do
    /* Keep key-value pairs until all attributes of this entity are
       processed.                                                        */
    properties ← ∅;
    foreach (a ∈ e.getAttributes()) do
        if a is a simple data type attribute then
            /* Add the value as a simple property value to the set of
               key-value pairs.                                          */
            properties ← properties ∪ {(a.getName(), a.getValue())};
        else if (a is a collection of a simple data type) then
            /* Add the collection of values as an array to the set of
               key-value pairs.                                          */
            properties ← properties ∪ {(a.getName(), a.getValuesAsArray())};
        else if (a is a named data type attribute) then
            /* Create an edge from the current entity to the entity
               referenced in the attribute value.                        */
            IRB.createEdge(e.getExpressId(), a.getName(), a.getValue(), ∅);
        else if (a is a collection of named data type attributes) then
            /* Create an edge for each reference also storing the
               position of the reference in the collection.              */
            i ← 0;
            foreach ref ∈ e.getValuesAsArray() do
                IRB.createEdge(e.getExpressId(), a.getName(), ref, {(position, i)});
                i ← i + 1;
            end
    end
    /* We have now processed all attributes of the entity and can
       therefore create the vertex in a single operation.                */
    IRB.createVertex(e.getExpressId(), properties);
end
Result: An internal graph representation (as described before in Section 4.2.3) of the input list of
        EXPRESS entities EX.
```

set. If it is not a simple data type attribute, but a **collection** of **simple data type** attributes, we transform the collection to an array of values and store this key-array pair in the properties set for later processing.

If the attribute is a **named data type** attribute or a collection of **named data type** attributes we can immediately create the edge in the property graph representation builder. We can do this, since references in *EXPRESS* themselves cannot have attributes and therefore we already know all the information we require to create the edge. In the case of a collection of **named data type** attributes we also number the edges in order to be able to reconstruct the order of the references later on.

After we have finished processing the last attribute of the *EXPRESS* entity, we are sure that we have found all attributes that should be converted to vertex properties. Therefore, we can now

create the vertex that represents the current entity. We repeat this process for every *EXPRESS* entity that is part of the IFC model.

**Remark 9.1**
We let the parser library handle the validation of the correctness of the input IFC data. This also includes that referential integrity is not violated, as we assumed before. □

## 9.3. Storing Multiple Building Models

Since we want to be able to store multiple buildings in our information system (see Section 7.1), we need to modify the schema presented in Section 3.2. Our new schema is depicted in Table 9.10 and modifications are highlighted in red:

Graphs : {[
GID : INT,
IFC_Header : VARCHAR
]}

Vertices : {[
GID : INT *(→ Graphs.GID)*,
VID : LONG,
Attributes : JSON
]}

OutgoingAdjacency : {[
GID : INT *(→ Graphs.GID)* ,
VID : LONG *(→ Vertices.VID)*,
$EID_1$ : LONG[],
$Label_1$ : VARCHAR,
$TID_1$ : LONG[],
...
$EID_k$ : LONG[],
$Label_k$ : VARCHAR,
$TID_k$ : LONG[]
]}

Edges : {[
GID : INT *(→ Graphs.GID)*,
EID : LONG,
SID : LONG *(→ Vertices.VID)*,
TID : LONG *(→ Vertices.VID)*,
Label : VARCHAR,
Attributes : JSON
]}

IncomingAdjacency : {[
GID : INT *(→ Graphs.GID)*,
VID : LONG *(→ Vertices.VID)*,
$EID_1$ : LONG[],
$Label_1$ : VARCHAR,
$SID_1$ : LONG[],
...
$EID_p$ : LONG[],
$Label_p$ : VARCHAR,
$SID_p$ : LONG[]
]}

Table 9.10.: The relational property graph database schema.

We introduce an additional table *Graphs* that stores metadata about the different graphs (or building models) in the database. For example, this includes the IFC file header. Next, we modify all tables to have an additional column that assigns each vertex and edge to the graph they belong to.

Using this altered schema we achieve the following properties:

1. We can store multiple IFC models in the same system.
2. We can query single models using the graphs GID as previously described in Chapter 5.
3. We can create queries that access data from multiple models.

While this modification enables us to simultaneously store several building models, it comes with a number of drawbacks:

1. Even though the models and hence graphs are completely disjunctive, in order to import additional models the existing tables and indices have to be modified. In particular, indices and constraints on the database schema significantly slow down the import of huge amounts of data. We could avoid this problem by temporarily deleting the indices and creating them after the data is imported. Still, the size of the indices and therefore their creation time will increase with each additional model and hence each successive model import will become slower.
2. Deleting single models comes with the same challenge as adding new models. We could avoid this by marking models as deleted and perform a cleanup task during times of low system load.
3. Although all queries from our use case only target single models (see Section 10.2.2), each query has to access the tables (or indices) containing all models. The increasing data size will therefore also slow down queries on single models.

Because the building models are disjunctive, we have a natural segmentation into the different available models. Leveraging this property of the graph, we can apply the concept of **horizontal fragmentation** well-known from distributed databases (e.g. see [ÖV20; Zha93]). We horizontally fragment the *Graphs* along the GID and apply the concept of derived fragmentation to the vertex and edge tables and store these locally in our database. We can use widely available features of database managements systems to automatically make our queries target the correct tables.

**Example 9.11**
Let us assume that our current database instance is not fragmented yet and contains multiple building models, including the models with the $GID = 223$ and $GID = 226$ as depicted in Table 9.12. For the sake of brevity we only present the vertex table as an example, the other tables are fragmented analogously.

| **Vertices** | | |
|---|---|---|
| **GID** | **VID** | **Attributes** |
| **223** | **6447** | **"Name": "Rathaus", "Type": "IfcBuilding", ...** |
| **...** | **...** | **...** |
| **226** | **6447** | **"Name": "Bauhof", "Type": "IfcBuilding", ...** |
| **226** | **7158** | **"Name": "OG 1 - Ebene 3", "Type": "IfcBuildingStorey", ...** |
| **..** | **...** | **...** |

Table 9.12.: Example vertex table before applying horizontal fragmentation.

We now derive the fragmentation along the GID. This results in the tables depicted in Table 9.13. Each of the two resulting tables only contains vertices of a single graph and the table contains all vertices of the graph. □

| **Vertices**$_{223}$ | | |
|---|---|---|
| **GID** | **VID** | **Attributes** |
| **223** | **6447** | **"Name": "Rathaus", "Type": "IfcBuilding", ...** |
| **...** | **...** | **...** |

| **Vertices**$_{226}$ | | |
|---|---|---|
| **GID** | **VID** | **Attributes** |
| **226** | **6447** | **"Name": "Bauhof", "Type": "IfcBuilding", ...** |
| **226** | **7158** | **"Name": "OG 1 - Ebene 3", "Type": "IfcBuildingStorey", ...** |
| **..** | **...** | **...** |

Table 9.13.: Two of the tables resulting from horizontally fragmenting the table depicted in Table 9.12 derived from the graph identifier GID.

By applying this horizontal fragmentation we overcome the drawbacks described before:

1. New IFC models are stored as a separate graph and will therefore be stored in their own set of tables. Hence, old indices do not have to be altered in order to insert the new data, but new indices for the new tables will be created and only the new data has to be considered. This also means that the import of new data will not be influenced by the amount of already imported models.

2. Deleting models from the IFC store is as simple as deleting the entry from the *Graphs* table and dropping the corresponding set of tables.

3. By using the well-known mechanisms for fragmentation, the query optimizer will only include relevant tables in the query evaluation. Therefore, if a single IFC model is the target of a query, only the tables containing data for this individual model (and the according indices) have to be loaded. It is still possible to include several or all models as targets of a query, which will be handled automatically by the query optimizer.

Overall we have found a way to simultaneously store numerous IFC models in the IFC store. The efficiency of queries performed in the system is not hindered by the number of models, but mainly influenced by the complexity of the target models of the query. For a detailed evaluation of the performance for up to 20 different models simultaneously see Chapter 10.

**Remark 9.2**
Our prototypical implementation uses the table inheritance mechanism [OH12] combined with check constraints available in **PostgreSQL** to implement local horizontal fragmentation.

By using this mechanism, the **PostgreSQL** optimizer is able to exclude tables that do not contain vertices or edges of the currently accessed graph, whenever we define queries with the filter WHERE $GID = x$.

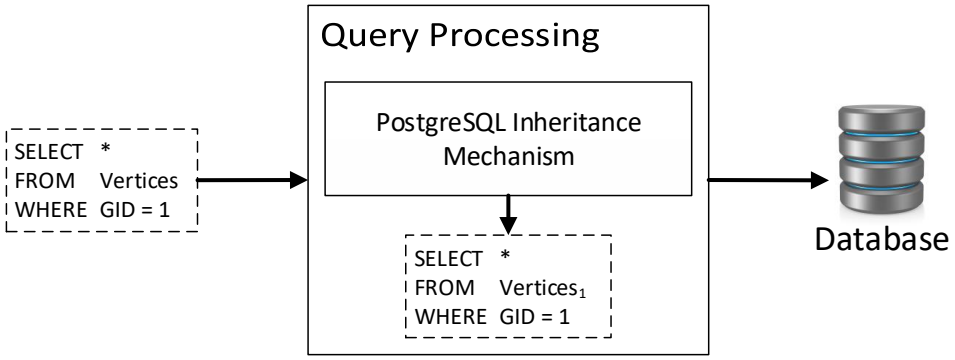



Figure 9.14.: Query processing for horizontally fragmented tables as depicted in Table 9.13.

The query processing for a query targeting a single graph is depicted in Figure 9.14: **PostgreSQL** automatically translates the query that targets a GID into a new version that only targets the relevant tables. Therefore, only the graph data that is necessary to answer the query has to be loaded.

□

## 9.4. Spatial Queries on the Building Model

So far we have presented how we store IFC building models in a RDBMS and given a general idea how the structural and semantical data can be queried using the approach presented in Chapter 5. But the data retrieved from an IFC file contains only the information the creator of the model included in the CAD software and also differs between different BIM software.

For example, the structural connection between two *IfcWall*s or the containment relation between an *IfcWall* and an *IfcWindow* does not have to be included in the data. At the same time, the geometric complexity of the building models can be very high. Therefore, direct spatial computation within a database in an interactive application context to retrieve this kind of information is not feasible.

Therefore, we abstract from the detailed geometric data and propose an approach to use GIS functionality and bounding boxes. These features are also widely available in most Relational Database Management Systems. To this purpose Bay [Bay18] created a prototypical implementation using PostGIS [Mar15].

The modified schema using PostGIS is depicted in Table 9.15: We add a table *IfcProductBoundingBoxes*, because only a few vertices of the large amount of vertices in the graph have a graphical representation. More specifically, only vertices representing *IfcProduct* entities can have a geometric representation assigned. These usually only account for a few hundred to thousands of vertices in a several hundred thousand vertex graph.

Bay compared the use of the *GeoJSON* data type, the *Geometry* data type and the *Box3D* data type to store bounding box information. The use of *GeoJSON* was an interesting approach, because using this format would not require a schema alteration. Nevertheless, Bay's work showed that using the *Geometry* data type is by far the most efficient for 3D GIS queries. For the case of the *ST_3DFullyWithin* operation provided by **PostGIS**, execution times using the *Geometry* approach were up to 200 times faster than the *GeoJSON* storage and up to 5 times faster than the *Box3D* approach [Bay18].

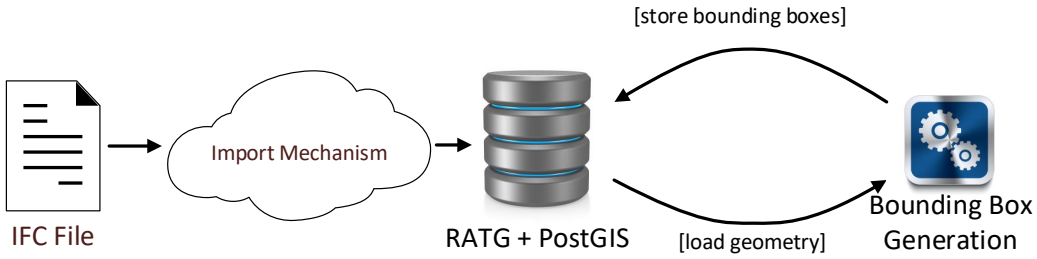



Figure 9.16.: The bounding box generation concept for **RATG**.

Our concept for the bounding box generation is depicted in Figure 9.16: We wanted to make the process independent of the import process, since not every instance of our use case

Graphs : {[
GID : INT,
IFC_Header : VARCHAR
]}

Vertices : {[
GID : INT *(→ Graphs.GID)*,
VID : LONG,
Attributes : JSON
]}

IfcProductBoundingBoxes : {[
GID : INT *(→ Vertices.GID)*,
VID : LONG *(→ Vertices.VID)*,
BoundingBox : Geometry
]}

OutgoingAdjacency : {[
GID : INT *(→ Graphs.GID)* ,
VID : LONG *(→ Vertices.VID)*,
$EID_1$ : LONG[],
$Label_1$ : VARCHAR,
$TID_1$ : LONG[],
...
$EID_k$ : LONG[],
$Label_k$ : VARCHAR,
$TID_k$ : LONG[]
]}

Edges : {[
GID : INT *(→ Graphs.GID)*,
EID : LONG,
SID : LONG *(→ Vertices.VID)*,
TID : LONG *(→ Vertices.VID)*,
Label : VARCHAR,
Attributes : JSON
]}

IncomingAdjacency : {[
GID : INT *(→ Graphs.GID)*,
VID : LONG *(→ Vertices.VID)*,
$EID_1$ : LONG[],
$Label_1$ : VARCHAR,
$SID_1$ : LONG[],
...
$EID_p$ : LONG[],
$Label_p$ : VARCHAR,
$SID_p$ : LONG[]
]}

Table 9.15.: The relational property graph database schema extended with GIS functionality.

application will require this feature. If it becomes necessary in the future, this process could be performed in parallel to the import process and use *GPU* acceleration. Then after the graph data was inserted into the database, bounding box data can be added. The geometry data is extracted from the database and bounding boxes are computed. These bounding boxes are then stored in the database. The prototype was usually able to generate bounding box models of up to 7,000 bounding boxes within a few minutes.

An example bounding box model of the *Conference Center* model is depicted in Figure 9.18 and was generated from the building model depicted in Figure 9.17. In order to make different bounding boxes visually distinct the boxes are colored with random colors.

Using GIS functionality enables us to estimate relations like containment or whether two *IfcProduct* instances are connected on the fly whenever these are not explicitly stated in the model data. Our first results look very promising, but further research is required, therefore

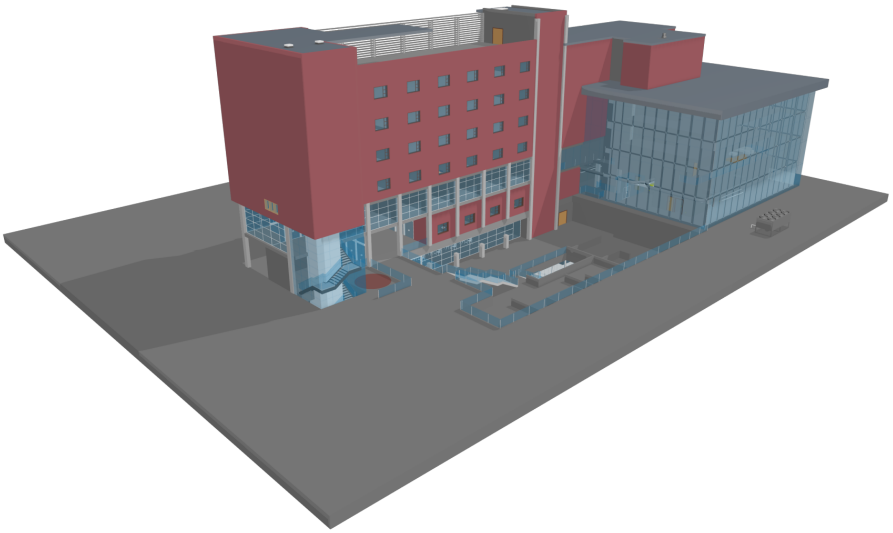


Figure 9.17.: A rendering of the Conference Center building model.

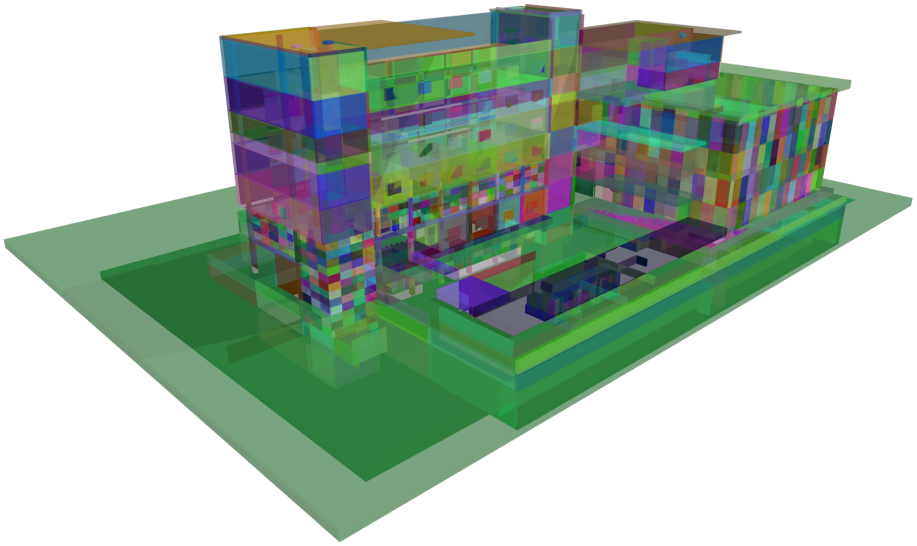


Figure 9.18.: A rendering of the bounding box representation of the building in Figure 9.17 taken from [Bay18].

we will not go into further detail in the scope of this work. Spatial queries will also not be part of our evaluation in the following chapter. For a more detailed description of our prototype we advise the reader to take a look at [Bay18].

# 10. Evaluation of the IFC Store

After having presented our use case in Chapter 7 and our approach to store IFC models as a property graph in Chapter 9, we now want to examine the performance of our approach. Therefore, we wanted to evaluate our storage approach for IFC data as we did with our general approach to store property graph data in a Relational Database Management System (RDBMS) (see Chapter 6).

We ran all tests on the same hardware as described in Chapter 6. The evaluation was conducted on a dedicated server with:

- two Intel Xeon 2.6GHz CPUs (in total 8 cores),
- 96 GB main memory,
- a 6 SSD RAID-0 (other RAID configurations would not have left enough storage capacity for the biggest data set),
- running 64-bit Ubuntu (version 18.04 LTS).

We used **PostgreSQL 13** on the aforementioned server, while we ran the client program of the benchmark on a (virtual) desktop computer that connected to the database over a 1Gbit/s LAN and both machines were connected to the same switch.

To evaluate the system on different building models, we used 25 different IFC models (a mixture of both IFC version 2x3 and 4) we had available ranging from 1MB to 280MB in size. The more complex models originate from tech demonstrations offered by various vendors and scientific institutes (and one from a future business partner), while the less complex models originated from our project partners. Since most of the models have been provided by project partners, we cannot publicly disclose the model files with this thesis. For the interested reader who wants to acquire the model files for their own test purposes, the author can forward any request for data release to the data owning party.

An overview of the IFC models we used for our use case evaluation is depicted in Table 10.2. The table is ordered by file size. Please note that a bigger file size cannot always be traced back to a more complex model. A major factor for the file size of a model includes but is not limited to the complexity of the geometric representation of the model. However, for our use case the complexity of the structural and semantical information contained in the model is more interesting. Therefore, the two most complex models are *DBK20* and the *OTC-Conference Center*, while for example the *Clinic_MEP* model contains much less semantic information even though having more than double the file size.

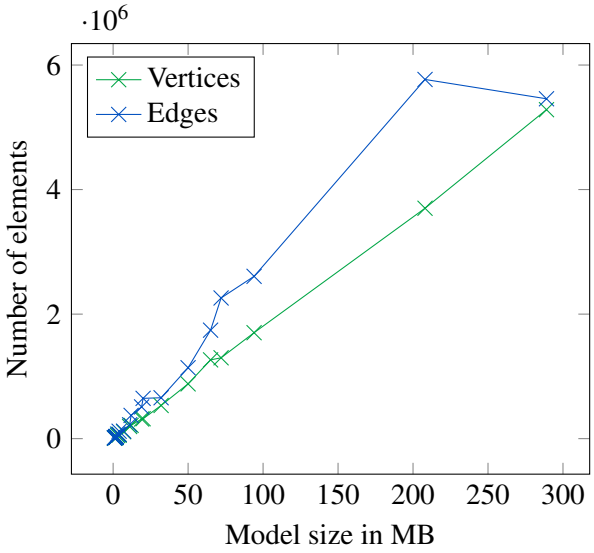



Figure 10.1.: Dependency between model size in MB and the number of vertices and edges in the model.

Taking a closer look at the data, we can see that the number of edges in the resulting graph is similar to the number of vertices (see e.g. Figure 10.1). Nevertheless, the queries that we require for our use case nearly always target vertices and edge types with a higher degree (for a detailed description of the queries see Section 10.2.2). With a few exceptions (e.g. *DC_Riverside_Bldg*) the relation between the number of vertices and number of edges is often predictable. We will see in the following evaluation that we can use the file size (or the number of vertices) to get a general idea of the time the import will require. In any case, the time required to import a model was lower than 10 minutes, which fulfills our use case's requirements.

| Model Name | Version | Size in MB | Nr. of Vertices | Nr. of Edges |
|---|---|---|---|---|
| Bauhof2 | IFC2x3 | 1MB | 6,074 | 7,605 |
| Kläranlage | IFC2x3 | 1MB | 12,190 | 13,381 |
| Kindergarten | IFC2x3 | 1MB | 12,429 | 15,363 |
| FWH-Gschwendt | IFC2x3 | 1MB | 12,923 | 15,130 |
| FWH-Ascha | IFC2x3 | 2MB | 19,598 | 22,139 |
| Bürgerhaus | IFC2x3 | 2MB | 22,495 | 25,943 |
| Grundschule | IFC2x3 | 2MB | 25,071 | 31,778 |
| Mehrzweckhalle | IFC2x3 | 2MB | 25,081 | 28,371 |
| Rathaus | IFC2x3 | 2MB | 28,059 | 35,187 |
| Vereinsheim-Sportheim | IFC2x3 | 2MB | 28,150 | 32,090 |
| AC20-FZK-Haus | IFC4 | 3MB | 48,786 | 72,801 |
| Mittelschule | IFC2x3 | 4MB | 56,525 | 64,785 |
| Domek-jednorodzinny | IFC2x3 | 4MB | 70,920 | 119,002 |
| Institute | IFC2x3 | 7MB | 110,805 | 120,692 |
| AC20-Institute-Var-2 | IFC4 | 11MB | 194,845 | 222,702 |
| DC_Riverside_STRUC | IFC2x3 | 12MB | 213,625 | 373,265 |
| Clinic_A | IFC2x3 | 19MB | 315,935 | 509,948 |
| Clinic_S | IFC2x3 | 20MB | 320,069 | 647,252 |
| Ettenheim-GIS | IFC2x3 | 32MB | 532,973 | 656,945 |
| Schependomlaan | IFC2x3 | 50MB | 879,530 | 1,138,890 |
| HITOS | IFC2x3 | 65MB | 1,265,059 | 1,742,271 |
| OTC-Conference Center | IFC4 | 72MB | 1,298,393 | 2,259,226 |
| DBK20 | IFC2x3 | 94MB | 1,701,674 | 2,606,837 |
| Clinic_MEP | IFC2x3 | 208MB | 3,700,014 | 5,768,342 |
| DC_Riverside_Bldg | IFC2x3 | 298MB | 5,283,000 | 5,458,041 |

Table 10.2.: All IFC models that were used to evaluate the suitability for our use case.

## 10.1. Import Performance

Our first prototypes to import the IFC data into a RDBMS took several hours for the *OTC-Conference Center* ($\sim$ 72MB). While this prototype enabled us to perform preliminary tests and a good estimation for the suitability of our approach, a runtime like this is still not acceptable for any practical application, even though it is only performed once for every building model.

Applying the general import concept presented in Chapter 4 to our use case as described in Section 9.2, we achieved a comparatively very fast import time of around 1.5 minutes for the same model. To get an impression of the import performance of our storage concept for IFC data, consider Figure 10.3.

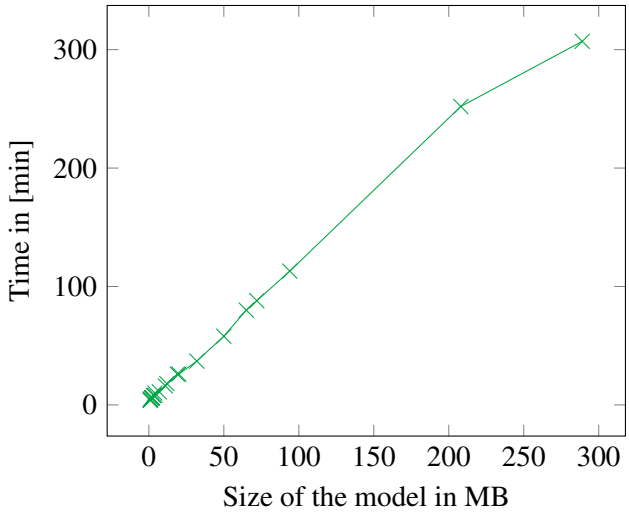



Figure 10.3.: Required time for the IFC data import depending on the size in MB.

Each import was performed several times and the mean import time is displayed, since the import times did never differ more than a second.

We know that the graphs that result from a IFC model contain a nearly linear number of vertices in relation to the file size of the building model. We are also aware that the resulting graphs are relatively thin. Therefore, the number of edges grows only slightly stronger than linear. From our general import performance evaluation using the LDBC-SNB benchmark data presented in Section 4.3 we know that the import time is in $\mathcal{O}(m\log(m))$ where $m$ is the number of edges.

Considering our findings depicted in Figure 10.3, our evaluation confirms that the import time mainly depends on the file size of the IFC model. Most of the building models we obtained from our project work could be imported in a few seconds, while even the biggest models (e.g. *DC_Riversid_Bldg*) was imported in less than 5 minutes. This is acceptable for our use

case, since the import of a building model into the information system is only performed a single time.

It is noted here that in contrast to other IFC stores like *BIMServer* [Bee+10b] (which required at least 1GB of main memory in order to load the biggest model), we only require 400MB of main memory to import any of the models, while most of this is due to a constant overhead that is necessary to initialize the IFC parsing mechanism.

## 10.2. Query Performance

To the best of our knowledge, there is no benchmark or comparable approach to evaluate IFC stores. There are several reasons for this: First, most IFC data servers use proprietary software and are closed systems that can only be used within the vendor software landscape. Second, there is no generally accepted query language or even query support for IFC servers, since the vendor's software solutions exchange the data in a closed and not accessible manner. Third, to the best of our knowledge no other IFC store is able to evaluate arbitrary queries on models or even across several models.

Consequently, we created our own evaluation setup to help us evaluate the approach tailored to our use case. As a proof of concept, Hartanto [Har20] created an IFC workload within the LDBC-SNB framework. We then built our own benchmark based on the queries we required for our *MonArch* use case. This will also enable us to compare our approach to future approaches. We will now present our approach to evaluating the suitability of our IFC store.

### 10.2.1. Methodology and Evaluation Setup

We used the LDBC-SNB benchmark framework and created our own workload within this framework. We can then use the LDBC-SNB framework to automatically initialize, execute, and measure the results for this workload. The general concept of our workload is depicted in Figure 10.4. The workload contains three major components:

**The set of IFC queries** Each of the query component represents a blueprint for a type of query that will be executed during a benchmark run. The query describes the parameters that have to be filled out in order to execute the query and contains the actual query that will be sent to the database. In addition, the IFC query skeleton also contains an *SQL* query that computes a list of parameters that can be filled in.

**The workload executor** The execution of queries is handled by the query executor. It also performs execution measurements. In order to execute a query, it automatically reads the set of parameters from parameter files. The parameters are read from files and not queried from the database for every execution in order to be able to make the workload executions repeatable, since we cannot guarantee that the queries that generate the parameters return those in the same order for every run.

**The parameter generator** In order to be able to create repeatable benchmark runs we can trigger the parameter generator to create a set of parameters. To do this, the parameter generator retrieves the queries that are provided by the query components and retrieves the set of parameters from the database. These are then stored in files that are repeatably used by the workload executor. This task is run as a preprocessing step of the benchmark execution and does not influence the actual results obtained using the benchmark.

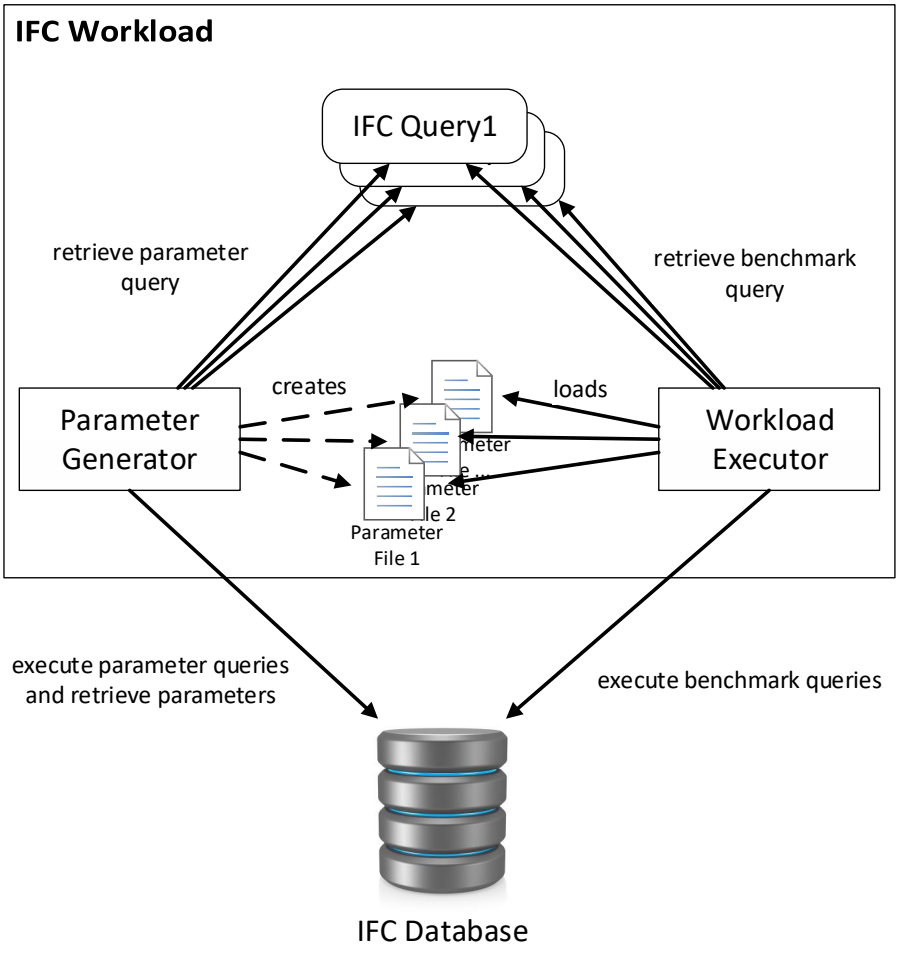



Figure 10.4.: The Industry Foundation Classes (IFC) workload concept.

The LDBC-SNB data set generator automatically generates a set of parameters for the queries of the LDBC-SNB workloads. In contrast, our IFC data is not generated, but was created in specialized CAD software. Therefore, we have to extract a set of parameters that we can use for the benchmark queries. We found that we can provide queries that can compute these sets of parameters from a given database instance. Within a few minutes we can extract the parameter from any number of models stored within the database. We verified this for database instances of up to 25 different IFC models.

This way we can automatically generate a set of parameters for a given database. This provides us a high level of flexibility regarding the data stored in the IFC database, since we can easily exchange the IFC models in the target database, generate a new set of parameters for this database instance and then perform a benchmark run.

**Remark 10.1**
The queries in our workload are all executed the same number of times, regardless of the distribution of occurrence in the real use case application, since we do not have data about that distribution from the real life application, yet. □

### 10.2.2. IFC Benchmark Queries

For now, we have identified 6 query types required for our use case. Since there is no standardized query language for IFC data, we use the property graph model and *Cypher* to describe the queries. The (simplified) *Cypher* descriptions of these queries are depicted in Listings 10.5 to 10.10.

**Complete Building Hierarchy (IFC1)** The first IFC query computes the complete building hierarchy starting at the *IfcBuilding* component. This query is implemented as a *recursive* SQL query that combines edges in *two different directions* and has to make sure that no cycles are returned. The building hierarchy is by far the most complex query that is required for our use case and is usually only computed once at application startup.

**Partial Building Hiearchy (IFC2)** The second IFC query computes the building hierarchy as did the first query, but does not start at the root node of the building. Therefore, this query starts at an arbitrary *IfcProduct* as a *recursive* SQL query that has to make sure, that the edges are traced in the *correct direction* (from complex to simple building elements) while using *different edge types* in *different directions*. We included this query in order to check if our approach would be suitable for lazy loading of the building hierarchy in the final application.

**Space Boundaries (IFC3)** The third IFC query searches for the bounding elements of a *IfcSpace*, like *IfcWall*, *IfcWindow* or *IfcDoor* elements. This can for example be used to find common walls between two rooms.

**Property Sets (IFC4)** The fourth IFC query retrieves *IfcPropertySets* that belong to an *IfcProduct* that contains *IfcSingleValue*s. This is the most used mechanism to store property sets that we have found for *IFC2x3* models in the set of models we encountered for our use case.

**Quantity Property Sets (IFC5)** The fifth IFC query we evaluated retrieves property sets that store quantity properties like volumes, lengths, or areas. Both the fourth and fifth query represent three edge-hop queries with defined edge labels. Our use case requires these queries in order to be able to display properties of building components.

**Windows in Walls (IFC6)** The sixth and last IFC query we wanted to evaluate retrieves all *IfcWindow*s (and additionally other elements like *IfcDoor*s) that are placed in a given *IfcWall*. This query represents a four edge-hop query. Our use case requires this query in order to display a more detailed and user group specific depiction of the building hierarchy and its components.

We used this set of query templates to evaluate the suitability of our approach for our given use case. We will now describe the application specific indexes we used in addition to the indices described in Section 3.2.1. Afterwards, we present our findings regarding the performance of our approach using the aforementioned queries.

```
MATCH (b:IfcBuilding)
    -[:RelatedElements
      |RelatedObjects
      |RelatingStructure
      |RelatingObject*]- (p)
RETURN b.GlobalId, p.GlobalId
```

Listing 10.5: IFC benchmark query 1: Compute the complete building hierarchy starting at the *IfcBuilding*.

```
MATCH (start:{GlobalId : ?})
    -[:RelatedElements
      |RelatedObjects
      |RelatingStructure
      |RelatingObject*]- (p)
RETURN start.GlobalId, p.GlobalId
```

Listing 10.6: IFC benchmark query 2: Compute the building hierarchy starting at an *IfcProduct*.

```
MATCH (space:{GlobalId : ?})
    <-[:RelatingSpace]- ()
     -[:RelatedBuildingElement]-> (b)
RETURN b.GlobalId
```

Listing 10.7: IFC benchmark query 3: Get the boundary *IfcProduct*s of an *IfcSpace*.

```
MATCH (product:{GlobalId : ?})
    <-[:RelatedObjects]- ()
     -[:RelatingPropertyDefinition]-> (propertySet:IfcPropertySet)
     -[:HasProperties]-> (property:IfcPropertySingleValue)
RETURN propertySet.name, property.name, property.value
```

Listing 10.8: IFC benchmark query 4: Get the *IfcSingleValue IfcPropertySet*s of an *IfcProduct*.

```
MATCH (product:{GlobalId : ?})
    <-[:RelatedObjects]- ()
     -[:RelatingPropertyDefinition]-> (propertySet:IfcPropertySet)
     -[:Quantities]-> (property)
WHERE (property:IfcQuantityVolume)
     OR (property:IfcQuantityLength)
     OR (property:IfcQuantityArea)
RETURN propertySet.name, property.name, property.value
```

Listing 10.9: IFC benchmark query 5: Get the *IfcQuantityPropertySet*s of an *IfcProduct*.

```
MATCH (wall:{GlobalId : ?})
    <-[:RelatingBuildingElement]- (:IfcRelVoidsElement)
     -[:RelatedOpeningElement]-> (:IfcOpeningElement)
    <-[:RelatingOpeningElement]- ()
     -[:RelatedBuildingElement]-> (filler)
RETURN filler.GlobalId, filler.name
```

Listing 10.10: IFC benchmark query 6: Get all *IfcWindow*s, *IfcDoor*s, etc. placed in an *IfcWall*.

### 10.2.3. IFC Application Specific Indexes

As we have described the application specific indexes for the LDBC-SNB evaluation in Section 6.2.2, we also additionally created indexes for our use case evaluation.

To obtain the best performance we could achieve for the IFC benchmark, we added the following application specific indices to the *vertex table*:

***Hash Index* on *GlobalId*** **IFC2** through **IFC6** all start at a given *IfcProduct* that must be retrieved using the *GlobalId*. In order to speed up the process of finding the starting point for the query, we add an index on the *GlobalId* using a hash index.

***Composite Index* on VID and *Type*** Some of the queries contain neighborhood queries that only search for a certain vertex *Type* of neighbors. Since this information is not stored in the *adjacency list* tables that we use to achieve this kind of query, a JOIN with the *vertex table* is required. Creating a composite B-tree index on the VID that is required to join the *vertex table* to the targets of the *adjacency list* and the *Type* attribute enables the DBMS to check the *Type* of the target vertex using an index-only-lookup. Therefore, the vertex data does not need to be loaded, which leads to a speedup of this type of queries.

We also added the following application specific indexes to the *edge list* table. Since the *edge list* table is only used for **IFC1** and **IFC2**, these indexes improved the performance of the two most complex queries:

***Composite Index* on SID and Label** In order to speed up queries that use graph patterns of variable length, we add a composite index on the source id SID and Label. More precisely, **IFC1** and **IFC2** both use graph patterns of variable length with undirected edges of a specified type.

***Composite Index* on TID and Label** Because **IFC1** and **IFC2** both use edge traversals in both directions, we also need to be able to efficiently traverse along edges in the opposite direction. Due to this, we also add a combined index on TID and Label.

No additional indexes have a positive effect on the benchmark execution performance for the *outgoing adjacency* table and the *incoming adjacency* table.

### 10.2.4. Query Performance Results

We evaluated our solution using the queries mentioned before by using the same approach as we did for evaluating **RATG** with the LDBC-SNB (see Chapter 6). We let the framework first perform several hundred warm-up queries, before the actual workload started. The actual workload consisted of 10,000 query executions that are roughly uniformly distributed across both the six different query types and the different building models that were imported in the IFC store.

Our goal was to verify the following two assumptions:

1. The approach answers queries fast enough for an interactive system that is used by several users at the same time. Since *IFC1* will be performed a single time at application startup and only in rare cases during usage of the application itself, *IFC1* should be answered in a few seconds. The rest of the queries needs to be answered in moments time, since the user will trigger these queries by selecting building parts in the application.

2. The throughput of the overall system is limited by the complexity of the most complex model and not by the number of models stored in the IFC store. This means that storing more models does not or barely influence the efficiency of queries that target a single building model.

We first imported two of the most complex building models we had available into the system and performed a benchmark execution. Then we successively added models that got lower in complexity. The throughput we achieved depending on the number of IFC models in the database is depicted in Figure 10.11.

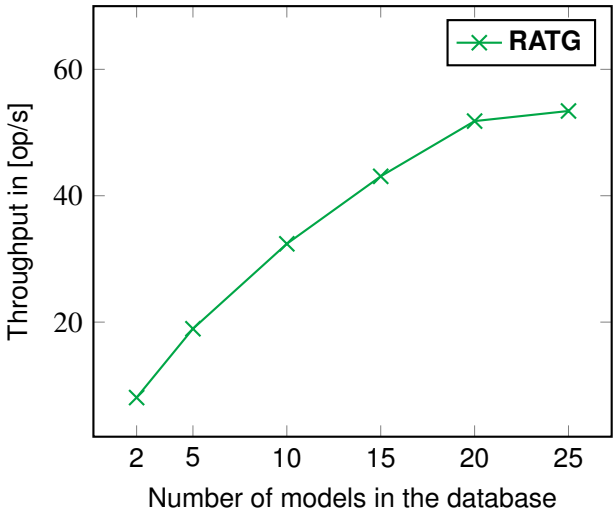



Figure 10.11.: Throughput of the IFC store depending on the number of different IFC models that have been imported.

We assumed that the most complex model in the system will limit the efficiency of the overall system due to the database design we described in Section 9.3. For our use case we expect the number of models in the system realistically to be up to a hundred models. This should not lead to any problems, since models that are currently not accessed do not have to be loaded by the database system due to the fragmentation design. Therefore, we expected that the throughput we can observe will not drop if we load additional models, which are not as complex as the models that are already in the system. Not only did the throughput not drop, but the throughput actually significantly increased when loading additional, less complex models. This is mainly due to the less complex building hierarchies present in the smaller models and a much faster evaluation of queries *IFC1* and *IFC2*.

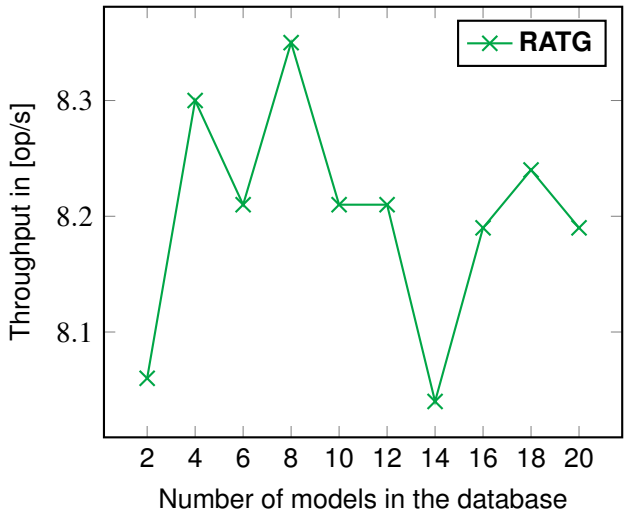



Figure 10.12.: Throughput of the IFC store depending on the number of times the same two models have been imported.

In order to further verify our assumption that a higher number of building models in the system will not negatively influence the performance of our approach, we applied the following approach: We chose the most complex building model we had and imported this building together with a second very complex model into the system. The second model was necessary, since the first model did not contain *IfcSpace*s and therefore *IFC3* would not have been executed. We then executed the workload on those two models. Afterwards, we successively imported those **same two models** again and ran the benchmark. The throughput while having those two models loaded 2 to 20 times into the system is depicted in Figure 10.12.

Once again, the findings confirm our assumption that additional (less or equally complex) building models will not influence the throughput of the overall system. All runs achieved a throughput between 8 and 8.4 operations per second with expectable small fluctations that are normal for a database system that is evaluated over an extended period of time.

Let us now take a look at the execution times of the six queries. We can split the six queries into two groups:

**IFC1 and IFC2** The first two queries result in complex recursive *SQL* queries that need to traverse incoming edges, as well as outgoing edges of vertices. In the use case application these two queries are either executed at application startup and the results cached in the client application, or run in the background to prefetch data that will be shown to the user in the near future. Therefore, the runtime requirements for those two queries are softer and a few seconds are easily acceptable.

**IFC3 through IFC6** The second group of queries will be triggered by user interaction in the client user interface. Therefore, fast response times for these types of queries are of utmost importance. A well-known threshold for response times that does not impact user productivity is 150 ms [TAS06]. Therefore, our goal is to be able to answer these types of queries in under 150 ms.

The runtime of the first two queries with the first 10 IFC models loaded into the system is depicted in Figure 10.13 in a boxplot diagram.

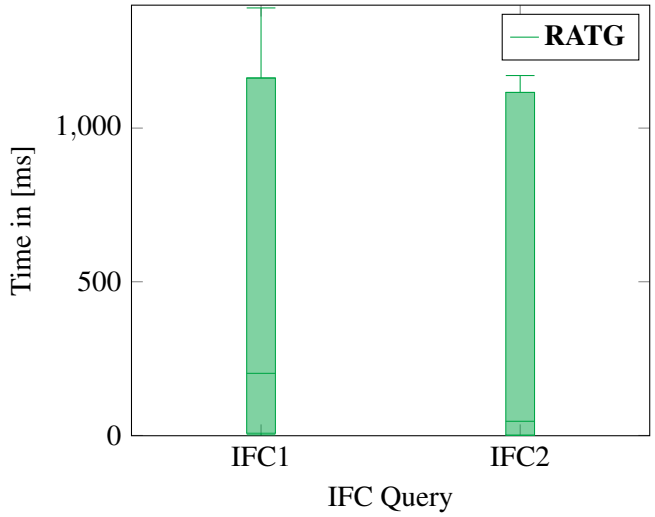



Figure 10.13.: Runtime of **IFC1** and **IFC2** on 10 different building models.

In contrast to the boxplot diagrams we used in Section 6.3, we chose the following configuration: The lower boundary of the box depicts the 25th percentile, the line in the box the 90th percentile and the upper bound of the box the 99th percentile. The two whiskers show the minimal and maximal execution times respectively.

Our findings show, that in 90% of query executions **IFC1** and **IFC2** return their results within 250ms and both return within $1,200$ ms in 99% of cases. In very rare exceptions the extraction of the building hierarchy (represented by **IFC1**) required 1.4 seconds. This only happens in rare executions on the most complex building model we had available. These execution times are perfectly suitable for our use case.

Our findings for **IFC3** through **IFC6** are depicted in Figure 10.14. Since these queries all require a few milliseconds, we chose to represent the 25th percentile as the bottom of the box, the 95th percentile as the line in the box and the 99th percentile as the top of the box. As before the whiskers represent the minimal and maximal execution times respectively.

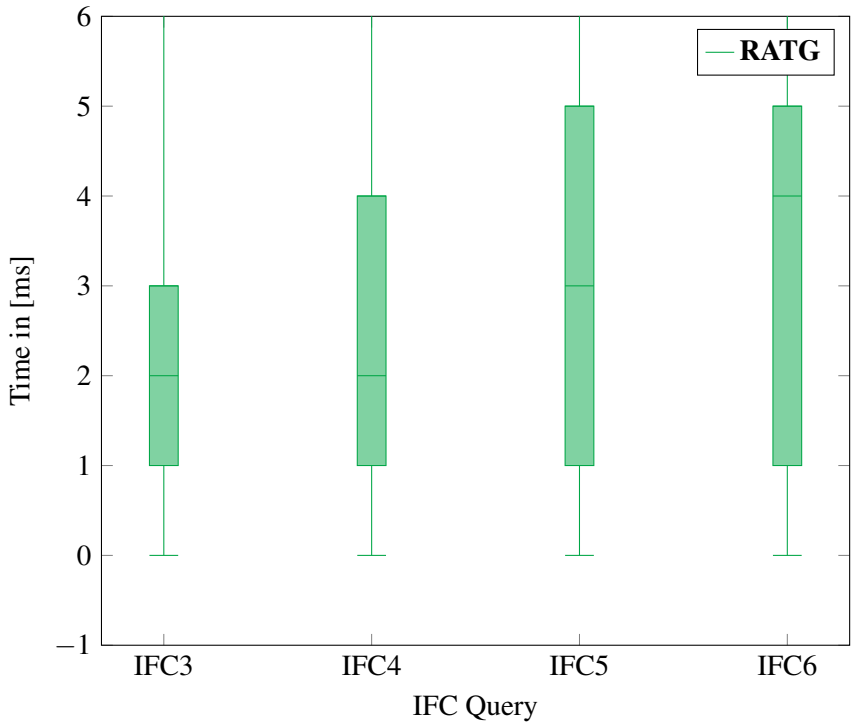



Figure 10.14.: Runtime of **IFC3** through **IFC6** of the IFC workload on 10 different building models.

As we can see, all interactively triggered queries are executed in under 5ms in 99% of cases. The absolute worst cases that occured in 10,000 query executions required just under 100ms to execute. As up to 150ms user productivity is not impacted by response time (see for example [TAS06]), even the worst case is more than appropriate for our application.

In conclusion, our evaluation on IFC data shows that our approach performs very well for the given use case. Especially for queries that will be interactively triggered by user interaction, the queries perform more than fast enough. Therefore, we have found a suitable solution to store the IFC building data of our use case in a Relational Database Management System.

# 11. Integration into the MonArch System

After having presented our approach to storing IFC data in a Relational Database Management System in Chapter 9 and having shown that the query performance we achieve is suitable for our use case, we will now present the integration of our IFC store into the *MonArch* system.

Our goal was to make the building data provided by the BIM methodology available in the *MonArch* System (see Figure 11.1): The building models are created in highly specialized CAD software, like for example Graphisoft Archicad[1].

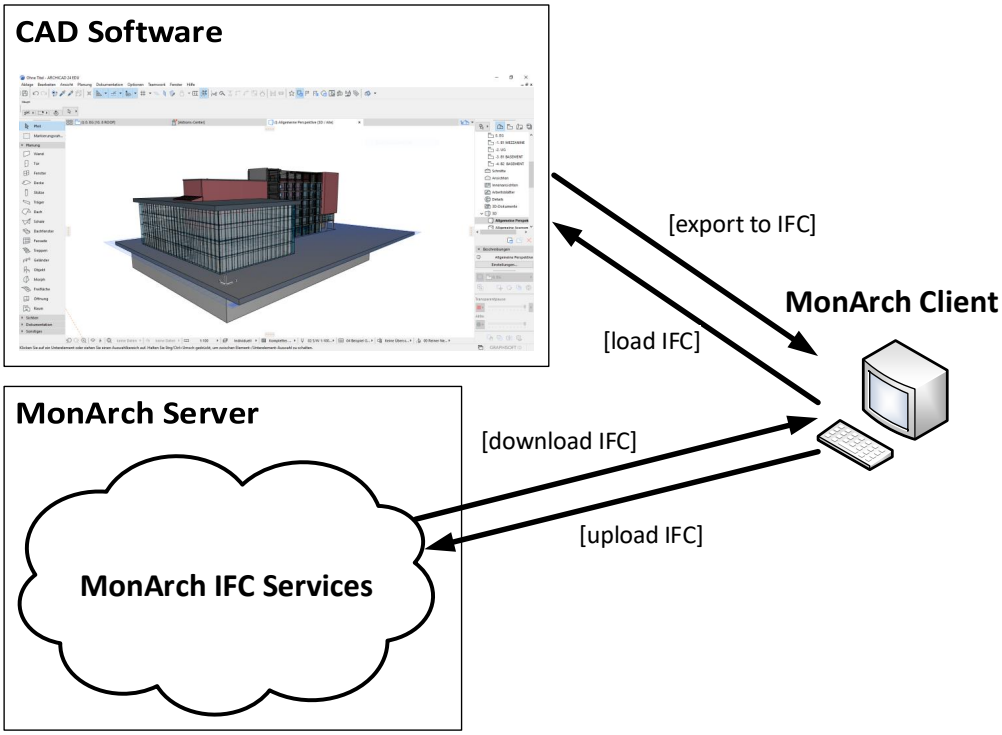



Figure 11.1.: The import/export cycle for IFC data in *MonArch*.

Because we did not want to depend on a specific software vendor, we adopt the OpenBIM approach and use the IFC data exchange format to export the data from the CAD software

[1] see https://graphisoft.com/de/archicad

and make it accessible for the *MonArch* client application. However, keep in mind that the *MonArch* client is not a BIM editor and the creation and modification of the building models themselves should take place in software specifically created for this purpose.

The *MonArch* client then sends the IFC file to the corresponding *MonArch IFC Services* located at the *MonArch Server*. The *MonArch* server now provides different services for accessing, modifying, and linking the building data with additional semantic information or simply linking digital documents to the building components.

```
ENTITY IfcProduct;
  ENTITY IfcRoot;
    GlobalId :  IfcGloballyUniqueId;
    OwnerHistory :  IfcOwnerHistory;
    Name         :  OPTIONAL IfcLabel;
    Description  :  OPTIONAL IfcText;
  ENTITY IfcObjectDefinition;
    [...]
  ENTITY IfcObject;
    [...]
  ENTITY IfcProduct;
    [...]
END_ENTITY;
```

Listing 11.2: The *EXPRESS* inheritance graph of the abstract class *IfcProduct* in *IFC4x2*.

Using the client, we want to be able to extract an IFC file that represents a building in the *MonArch*. This model can then be modified further in CAD software.

After importing the model into the *MonArch* system again, previously added information should still be available without the need to perform the linking again. We published a more detailed description of our concept for integrating the 3D building model data into the *MonArch* system in [SES19].

As we have previously described in Section 7.1, the main requirements for integrating the building data into the *MonArch* system with the aforementioned properties are:

1. We need to be able to segment the building into components. Since the BIM methodology itself is based upon a component based construction of the building model, the open exchange format IFC naturally also is segmented into building elements.

2. We must be able to identify each component in order to reference it within the *MonArch* system and attach additional semantic information. *IfcProduct* is the abstract superclass of the classes that represent any object that relates to a geometric or spatial context. Therefore, also all physical building components, as well as conceptual components like *IfcSpace*, are a subclass of *IfcProduct*. The *IfcProduct* (the inheritance hierarchy is depicted in Listing 11.2) class is derived from the abstract *IfcRoot* class. This class defines the **mandatory** attribute *GlobalId*, which is an IFC specific encoding

of a Universally Unique Identifier (UUID) [LMS05] and hence makes each physical building component identifiable. CAD software is supposed to import this UUID and not change it during modification of the model. We have verified this assumption for the two widely adopted CAD solutions *AutoDesk Revit* and *Graphisoft Archicad*.

Within the *MonArch* **PostgreSQL** instance we can then use a composite foreign key mechanism containing the GID and UUID (see Table 9.10 for a depiction of the database schema) to link additional information to the vertices in the IFC store.

We assume that these UUIDs are stable as defined in [ISO21]. Then we can simply exchange the GID component of the foreign key and redirect the reference to the newly imported graph in the system. Because the UUID did not change all references to IFC vertices now point to the correct vertex in the newly imported building model.

**Remark 11.1**
If the modification of an already imported building model deletes an *IfcProduct* we cannot just redirect the reference to the same building element. We will then redirect the pointer to the next higher element of the building hierarchy, which we can extract using the query depicted in Listing 10.6. □

## 11.1. Microservice Architecture for the IFC Integration

Since we wanted to seamlessly integrate our approach to storing IFC data into a RDBMS, we decided on a microservice architecture [Nad+16]. Our *MonArch* client application then acts as the coordinator and communicates with the microservices using the Representational State Transfer (REST) Application Programming Interface (API) [RR08].

Our decision of using the design of loosely-coupled services is based on a number of reasons and advantages:

1. The Industry Foundation Classes are rather extensive. Therefore, we use open-source software to parse the data and to generate the 3D data for rendering. New open-source software tools that support IFC data are developed and released constantly. The microservice architecture allows us to easily replace components that have a single task whenever better suited open-source software tools emerge.
2. IFC is still actively developed and extended. Again, the microservice architecture makes it possible to easily adapt single components to new versions of the IFC schema.
3. This type of architecture enables us to deploy different tasks to different machines. We can for example separate the generation of the 3D geometry and deploy it to a different machine than the IFC store for graph data. Then the computation-intensive task of generating the geometric 3D model can be performed on its own *GPU* machine and does not influence the performance of the rest of the *MonArch* system.
4. We can perform different tasks on different servers: While the expensive preparation for the IFC data import described in Section 9.2, the extraction of the 3D rendering model can be performed on a different machine while none of the processes impede each other. At the same time, the service offering query capability for already imported data is fully available.

Due to these reasons, we decided to split the integration of the IFC data into a set of microservices. Since the *MonArch* system follows a classical client-server architecture, the client application assumes the role of the coordinator for the first prototypical implementation. In future versions of the system the services themselves will start communicating between each other, for example in order to coordinate the order of the import of the graph data and the attachment of bounding box information to the vertices once the graph import has concluded.

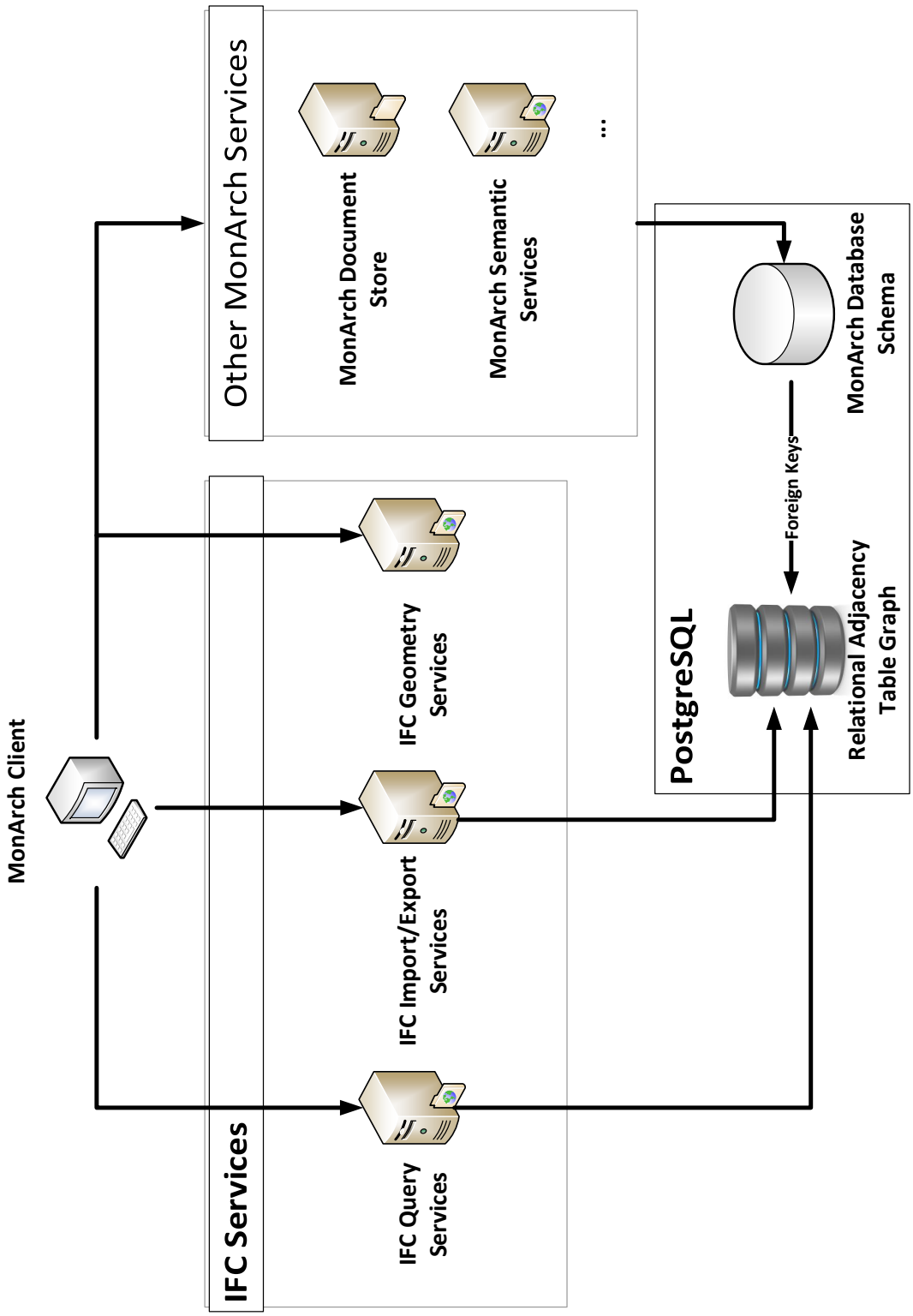



Figure 11.3.: The integration concept for integrating the IFC store in *MonArch*.

The complete microservice landscape and the components resulting from our integration of IFC data into the *MonArch* system is depicted in Figure 11.3. We use *Spring Boot* [Wal15] to implement the different server microservice components:

**IFC Import/Export Services (based on Chapters 4 and 9)** This service offers functionality to import and export IFC data to **RATG**. To do this it uses the concepts presented in Chapter 4 and Section 9.2. We use the BimServer [Bee+10b] parsing component to implement the parsing layer of the import algorithm. This is a prime example for the advantage of our microservice architecture: Unfortunately, while BimServer works well for older IFC versions it is not well maintained and still runs on *Java 9*. It does not work on current Java versions and we can not be sure if it will work with future IFC releases. Therefore, will replace this component in the near future.

**IFC Query Services (based on Chapters 5 and 10)** The query service offers an interface to perform the queries described in Section 10.2.2 and uses the translation concept presented in Chapter 5. Due to security measures, currently only predefined queries can be sent to this service. When more work has been we will offer a direct *Cypher* interface through this service.

**IFC Geometry Services (partly based on Section 9.4)** We use the open-source tool *IfcOpenShell* to generate the 3D rendering model [Kri]. *IfcOpenShell* creates a *Blender* [Mul11] model that contains each *IfcProduct* including its UUID. In future application releases this service will also contain the concepts presented in Section 9.4.

**Relational Adjacency Table Graph (based on Chapter 3)** The actual graph data is stored in its own schema within the *MonAarch* **PostgreSQL** instance. Therefore, the additional information stored in the *MonArch* database schema can reference the data in **RATG** using the standard RDBMS mechanisms.

**MonArch Client (not directly part of this work)** The client assumes the role of the coordinator. We implemented the prototypical client using the Eclipse Modelling Framework (EMF) [Ste+08] to create the client side model and generate most of the user interface. To render the 3D models we use jMonkeyEngine [Kus13].

In order to import an IFC file into the *MonArch* system, the client application sends the IFC file to the **IFC Import/Export Services**. While this service performs the import work, the client sends the IFC file to the **IFC Geometry Services**. The **IFC Geometry Service** generates a .blender file and sends it back to the client. When the import is finished, the client application sends the request to the **IFC Query Services** to extract the building hierarchy and caches the results and creates proxy objects in the *MonArch* schema. Whenever the properties of a building element are requested, the client uses the **IFC Query Services** to retrieve the property sets of the building component. In addition, the IFC file is stored in the **MonArch Document Store** and linked to the building root node for quick export of the original data.

Together with the original *MonArch* services the whole system is now able to integrate IFC data with arbitrary digital documents, semantic data (including open linked data [BVS18]) and GIS data.

## 11.2. The MonArch BIM Prototype

Finally, we created a prototypical application based on the research and concepts previously presented in this work in the context of the project *BaBeDo*.

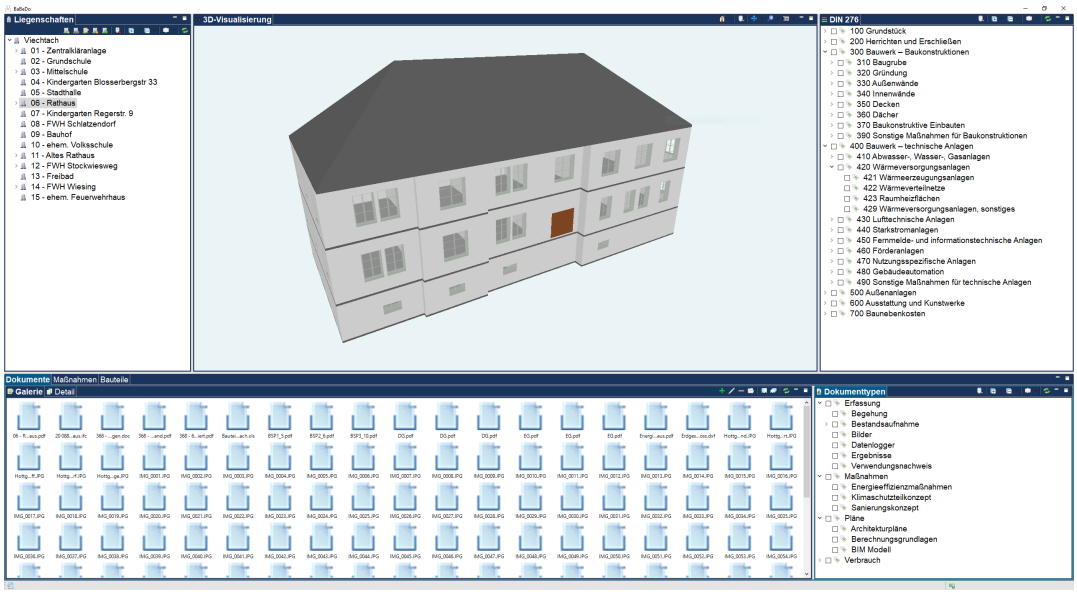



Figure 11.4.: A screenshot of the final prototype application for the IFC integration.

Figure 11.4 depicts a screenshot of the final prototype client:

- On the left-hand side, we see a list of the buildings documented in the system. All these buildings are imported BIM models using the aforementioned **IFC Import Services** microservice (see Section 11.1). The full hierarchy of each building can be viewed in a file explorer-like manner (can be seen in Figure 11.5) and is retrieved from the *MonArch* server component using the **IFC Query Services**.
- In the middle of the interface the *JMonkeyEngine* [Kus13] renders the graphical representation of the building. Because we can link all building components using the UUIDs provided by the model, a user can both select a building component in the component tree, as well as in the 3D model. The list of building components in the explorer-like hierarchy, the 3D rendering and an optional two dimensional navigation map are fully synchronized. Therefore, a selection in one of the three mechanisms results in a selection in the other two UI components. The 3D model is an addition to the previously available 2D navigation maps available in *MonArch* and neither are strictly necessary to use the system.
- On the right-hand side, two different semantic topic trees are visible and can freely be linked with the building components, as well as with the documents stored in the

system. The semantic linking of documents and building components is a feature of the original *MonArch* system.

- The documents that are available for the current selection of building elements and topics is visible at the bottom of the screenshot. Thereby, the selection of documents that is presented to the user is defined by the chosen building components as well as the topics chosen in the topic trees on the right-hand side.

Figure 11.5 depicts another view of the final prototype application:

- The middle of the application now shows the rendering of the property sets assigned to the building component chosen in the component tree on the left side. This propery set can be retrieved from the server component using the **IFC Query Service** microservice presented in Section 11.1.
- On the right top side the GIS component of the *MonArch* system is visible. Each building can be located on an *Open Street Map* [Ben10] rendering. The selection mechanism on the location markers is also synchronized with the building tree, the 3D rendering, and the 2D navigation map and can therefore also be used to select the desired building.

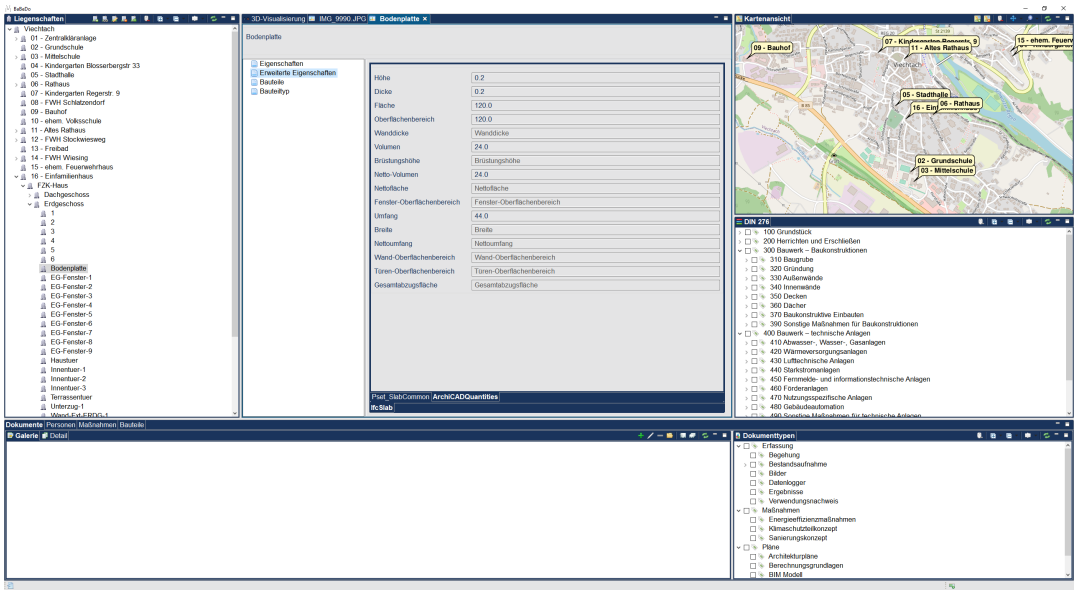


Figure 11.5.: A screenshot of the final prototype application displaying a property set retrieved from **RATG**.

Project *BaBedo* and therefore part of this research was supported by the "Bayrische Staatsministerium für Wirtschaft, Energie und Technologie" in the context of "BaBeDo" (IUK568/001) which we conduct in partnership with *VEIT Energie GmbH*.[2].

[2]https://www.veit-energie.de/

# Part IV.

# Summary and Conclusion

# 12. Summary

In this chapter we summarize the main aspects and results of this thesis and discuss possible future work.

## 12.1. Future Work and Outlook

Many applications for graph databases are not based on writing declarative queries, but need to use APIs. In consideration of this requirement, Nguetsa [Dom20] has developed a prototype for a API for **RATG**. While Nguetsa's approach already considers prefetching strategies to improve the efficiency of the API, more research is needed to make the approach viable in comparison to native graph databases.

Our approach to store property graph data, **Relational Adjacency Table Graph** (**RATG**), could be enhanced by further improvement of the *Cypher* mechanism. Gojayev [Goj21] developed a concept to integrate query optimization using a dynamic programming approach. Hereby, only the selectivity of targeted vertices were considered. Enhancing upon Gojayev's approach by also considering the selectivity of e.g. edge types could improve query optimization for our system.

Shanmugam [Sha21] started to complete the translation mechanism for *Cypher* queries with features of *Cypher* that had not been considered, yet. Currently, spatial queries using the GIS capability of the RDBMS have to be written manually. Integrating GIS query functionality into *Cypher* and the translation mechanism, would further increase the usability of our approach.

Vogt [Vog19] has already started to work on a view concept for the IFC graph. Further work in this area could enable us to restrict queries on predefined views of the graph or even to export parts of the building. Discussions with users of the *MonArch* system have shown that especially functionality to export parts of a building are of interest for possible use cases, since this would enable us to provide limited data access to different stakeholders of a building.

Finally, by getting more user data from our applications we could make our workload for the IFC store more realistic and increase its suitability to compare our IFC store to other future solutions.

## 12.2. Conclusion

Motivated by our building information system use case ***MonArch*** (see Chapter 1), we found a way to store **IFC building models** in a **RDBMS** using a **transformation** of the building data into the **property graph model**. Hence, we extensively investigated the **storage** of **property graphs** in **Relational Database Management System**.

To this end, we surveyed related work regarding the storage, query possibilities and existing means to evaluate property graphs in RDBMS (see Chapter 2).

Afterwards, in Chapter 3 we defined the property graph model used for this work and introduced our novel database schema **Relational Adjacency Table Graph** (**RATG**). Our approach leverages the widely-available database features to store and query **JSON** data in relational tables and hence **circumvents** the restrictions of the **fixed number of columns** to store vertex and edge properties. We also use **redundant storage of edge data** in **adjacency lists**, shredding different edge labels into a fixed set of column triples. This is done by applying **hash functions** to edge labels and so manages in most cases to store the complete adjacency of a vertex within a single tuple. We also gave an overview of how to use the schema for querying data in a property graph manner.

Then we presented the concept for importing huge data sets into **RATG** in Chapter 4. We have presented a **novel algorithm** to **compute hash functions** based on *UML*-like **data models**. Using these hash functions, we described an **algorithm to efficiently import huge data sets** into the database. Using this algorithm, we are able to load data sets into the database with the size of the finished database exceeding the main memory by more than factor 10.

Since *SQL* queries targeting **RATG** become rather massive in shear text size, we presented an **approach to integrate** the intuitive property graph query language ***Cypher*** into our system (see Chapter 5). To this end, we presented the syntax, the semantics, and a **formal translation mechanism** to convert *Cypher* queries into their *SQL* equivalent, which can then be executed on the RDBMS.

Furthermore, in Chapter 6 we evaluated the efficiency of **RATG** using the standardized benchmark framework Linked Data Benchmark Council - Social Network Benchmark (LDBC-SNB) and shown the validity of our approach to store general property graph data.

After having focused on the storage of general property graph data in the first part of this work, we then took a closer look at **RATG** for our **use case**. Hence, in Chapter 7 we presented our use case in more detail. To this end, we have described the data model of the Industry Foundation Classes (IFC), which are the basis for our 3D building data integration.

We then described related work regarding the storage and query capability of the few available open IFC store approaches and the concepts to interpret an IFC model as a graph in Chapter 8.

Afterwards, in Chapter 9 we described our **approach to storing IFC data** in the property graph schema **RATG** in detail. We have **defined a mapping** from an *EXPRESS* data model to the property graph model and described an **algorithm to perform this conversion**. We

also showed, how we can **store multiple building models** within the same **RATG** instance without losing import efficiency or query performance on single models, as well as a concept to **enable spatial queries** on the building models using GIS features of the RDBMS.

After having described our concept to store IFC data in our property graph schema, we then presented our approach to **evaluating IFC model stores** in Chapter 10. We integrated our own **novel workload** into the LDBC-SNB benchmark framework. Using this workload, we can evaluate the IFC store using queries directly required by our use case *MonArch*. We have been able to show that even in the absolute worst case we encountered during our evaluation, our approach **performs exceedingly well** and enables the integration of IFC data into *MonArch* and can smoothly be used interactively.

Finally, in Chapter 11 we described our **integration** of the developed IFC model store **into the *MonArch* system** using a **microservice architecture** and showed how this integration looks in the client application prototype.

In conclusion, we have found a highly suitable way to integrate BIM building data into the *MonArch* system using the IFC exchange format by converting the data into a property graph model and storing this within the RDBMS.

# A. Detailed Evaluation Results for the LDBC-SNB

The following tables show the results we achieved using the LDBC-SNB. We disabled *CQ14*, which is displayed by "-" instead of a value. ↯ symbols mark executions that did not terminate, and we therefore got no results from the benchmark framework.

| | LDBC-SNB SF1 | | | LDBC-SNB SF3 | | | LDBC-SNB SF10 | | |
|---|---|---|---|---|---|---|---|---|---|
| Query | 25%-Quantile | Median | 75%-Quantile | 25%-Quantile | Median | 75%-Quantile | 25Quantile–index7 | Median–index8 | 75Quantile–index9 |
| CQ1 | 366 | 396 | 429 | 1010 | 1060 | 1097 | 7089 | 7198 | 7428 |
| CQ2 | 1363 | 1448 | 1509 | 3593 | 3721 | 3856 | 10964 | 11563 | 12157 |
| CQ3 | 5034 | 5349 | 5716 | 13844 | 14854 | 15704 | 7310 | 8422 | 10223 |
| CQ4 | 158 | 239 | 417 | 939 | 1761 | 3417 | 968 | 1706 | 5362 |
| CQ5 | 1698 | 2305 | 4530 | 3313 | 5417 | 11369 | 1656 | 1892 | 2995 |
| CQ6 | 16073 | 16407 | 19069 | 85572 | 96408 | 106556 | 381472 | 499760 | 538848 |
| CQ7 | 179 | 305 | 492 | 11 | 11 | 12 | 11 | 12 | 17 |
| CQ8 | 4307 | 4567 | 4662 | 17536 | 17742 | 18015 | 53872 | 55560 | 56580 |
| CQ9 | 17438 | 19336 | 21657 | 48850 | 49228 | 54130 | 73952 | 75824 | 77896 |
| CQ10 | 2272 | 2298 | 2344 | 4783 | 6233 | 8324 | 10957 | 12654 | 19250 |
| CQ11 | 1345 | 1380 | 1418 | 343 | 373 | 403 | 830 | 897 | 1003 |
| CQ12 | 1189 | 1299 | 1470 | 1907 | 2097 | 2271 | 4889 | 6277 | 7133 |
| CQ13 | 216 | 248 | 264 | 570 | 636 | 673 | 6697 | 7023 | 7253 |
| CQ14 | - | - | - | - | - | - | - | - | - |
| SQ1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 2 |
| SQ2 | 2235 | 2266 | 2302 | 9158 | 9257 | 9367 | 30480 | 31031 | 31691 |
| SQ3 | 1 | 2 | 2 | 1 | 1 | 2 | 2 | 3 | 4 |
| SQ4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| SQ5 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| SQ6 | 2215 | 2245 | 2275 | 9124 | 9214 | 9306 | 30294 | 30790 | 31298 |
| SQ7 | 10 | 13 | 2246 | 9 | 14 | 9218 | 30646 | 34722 | 61384 |
| UQ1 | 12 | 12 | 12 | - | - | - | | | |
| UQ2 | 2 | 3 | 3 | 2 | 3 | 3 | 4 | 4 | 5 |
| UQ3 | 2 | 2 | 3 | 2 | 3 | 3 | 4 | 4 | 5 |
| UQ4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 6 | 10 |
| UQ5 | 2 | 2 | 3 | 2 | 2 | 3 | 3 | 4 | 5 |
| UQ6 | 4 | 4 | 5 | 4 | 4 | 6 | 5 | 8 | 11 |
| UQ7 | 5 | 6 | 8 | 6 | 8 | 19 | 7 | 8 | 13 |
| UQ8 | 2 | 2 | 3 | 2 | 2 | 2 | 3 | 3 | 4 |

Table A.1.: All SF 1-10 Results for **SQLGraph**.

| | LDBC-SNB SF30 | | | LDBC-SNB SF100 | | |
|---|---|---|---|---|---|---|
| Query | 25%-Quantile | Median | 75%-Quantile | 25Quantile–index4 | Median–index5 | 75Quantile–index6 |
| CQ1 | 41508 | 49764 | 62320 | ↯ | ↯ | ↯ |
| CQ2 | 62668 | 70828 | 76248 | ↯ | ↯ | ↯ |
| CQ3 | 875392 | 1272832 | 1869952 | ↯ | ↯ | ↯ |
| CQ4 | 24358 | 33206 | 41122 | ↯ | ↯ | ↯ |
| CQ5 | 286576 | 396240 | 489168 | ↯ | ↯ | ↯ |
| CQ6 | 3724800 | 4006400 | 4617216 | ↯ | ↯ | ↯ |
| CQ7 | 46 | 66 | 92 | ↯ | ↯ | ↯ |
| CQ8 | 215856 | 262496 | 301008 | ↯ | ↯ | ↯ |
| CQ9 | 520672 | 1293632 | 1396288 | ↯ | ↯ | ↯ |
| CQ10 | 55526 | 60370 | 70068 | ↯ | ↯ | ↯ |
| CQ11 | 7583 | 8905 | 9655 | ↯ | ↯ | ↯ |
| CQ12 | 59020 | 77448 | 88432 | ↯ | ↯ | ↯ |
| CQ13 | 20210 | 30847 | 47106 | ↯ | ↯ | ↯ |
| CQ14 | - | - | - | ↯ | ↯ | ↯ |
| SQ1 | 2 | 3 | 4 | ↯ | ↯ | ↯ |
| SQ2 | 95080 | 97888 | 101208 | ↯ | ↯ | ↯ |
| SQ3 | 7 | 16 | 38 | ↯ | ↯ | ↯ |
| SQ4 | 1 | 1 | 1 | ↯ | ↯ | ↯ |
| SQ5 | 1 | 2 | 2 | ↯ | ↯ | ↯ |
| SQ6 | 252896 | 258048 | 264896 | ↯ | ↯ | ↯ |
| SQ7 | 96392 | 106664 | 194040 | ↯ | ↯ | ↯ |
| UQ1 | - | - | - | ↯ | ↯ | ↯ |
| UQ2 | 5 | 6 | 8 | ↯ | ↯ | ↯ |
| UQ3 | 5 | 6 | 7 | ↯ | ↯ | ↯ |
| UQ4 | 7 | 9 | 10 | ↯ | ↯ | ↯ |
| UQ5 | 5 | 6 | 8 | ↯ | ↯ | ↯ |
| UQ6 | 4 | 5 | 7 | ↯ | ↯ | ↯ |
| UQ7 | 9 | 12 | 18 | ↯ | ↯ | ↯ |
| UQ8 | 4 | 5 | 6 | ↯ | ↯ | ↯ |

Table A.2.: All SF 30-100 Results for **SQLGraph**.

| | LDBC-SNB SF1 | | | LDBC-SNB SF3 | | | LDBC-SNB SF10 | | |
|---|---|---|---|---|---|---|---|---|---|
| Query | 25%-Quantile | Median | 75%-Quantile | 25%-Quantile | Median | 75%-Quantile | 25Quantile–index7 | Median–index8 | 75Quantile–index9 |
| CQ1 | 600 | 627 | 662 | 903 | 942 | 979 | 1712 | 1806 | 1885 |
| CQ2 | 393 | 410 | 432 | 353 | 372 | 392 | 427 | 447 | 453 |
| CQ3 | 4932 | 5307 | 5484 | 11418 | 12201 | 12649 | 8748 | 8796 | 9994 |
| CQ4 | 580 | 861 | 1657 | 4852 | 12325 | 59242 | 4332 | 11550 | 21706 |
| CQ5 | 1559 | 2080 | 4085 | 3746 | 5885 | 11701 | 1544 | 1718 | 3209 |
| CQ6 | 11840 | 13360 | 15697 | 53640 | 66896 | 67048 | 352576 | 370336 | 391136 |
| CQ7 | 170 | 269 | 422 | 3 | 5 | 6 | 4 | 8 | 9 |
| CQ8 | 4 | 4 | 5 | 154 | 198 | 229 | 3 | 4 | 5 |
| CQ9 | 8491 | 8872 | 9627 | 16883 | 18322 | 19193 | 34152 | 39364 | 39364 |
| CQ10 | 3543 | 3635 | 3695 | 1404 | 1507 | 1588 | 3282 | 3604 | 4333 |
| CQ11 | 459 | 485 | 503 | 13109 | 14812 | 16177 | 3345 | 10038 | 16339 |
| CQ12 | 976 | 1110 | 1209 | 1198 | 1310 | 1449 | 3536 | 4252 | 7158 |
| CQ13 | 172 | 191 | 205 | 435 | 483 | 513 | 4735 | 5040 | 5215 |
| CQ14 | 150 | 150 | 150 | 150 | 150 | 150 | 150 | 150 | 150 |
| SQ1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| SQ2 | 7 | 16 | 32 | 2389 | 2514 | 2577 | 744 | 809 | 886 |
| SQ3 | 1 | 1 | 2 | 1 | 2 | 2 | 1 | 2 | 3 |
| SQ4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SQ5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SQ6 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| SQ7 | 79 | 80 | 84 | 0 | 0 | 1 | 0 | 1 | 2 |
| UQ1 | 12 | 12 | 12 | - | - | - | - | - | - |
| UQ2 | 2 | 3 | 3 | 2 | 3 | 3 | 4 | 4 | 6 |
| UQ3 | 2 | 3 | 3 | 2 | 2 | 3 | 3 | 4 | 5 |
| UQ4 | 5 | 6 | 6 | 5 | 7 | 8 | 23 | 23 | 23 |
| UQ5 | 2 | 3 | 3 | 2 | 2 | 3 | 3 | 4 | 5 |
| UQ6 | 21 | 70 | 90 | 59 | 246 | 1298 | 152 | 218 | 317 |
| UQ7 | 27 | 81 | 415 | 70 | 234 | 1318 | 206 | 290 | 792 |
| UQ8 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 3 | 3 |

Table A.3.: All SF 1-10 Results for **RATG**.

| | LDBC-SNB SF30 | | | LDBC-SNB SF100 | | |
|---|---|---|---|---|---|---|
| Query | 25%-Quantile | Median | 75%-Quantile | 25Quantile–index4 | Median–index5 | 75Quantile–index6 |
| CQ1 | 3430 | 3884 | 4333 | 5399808 | 5399808 | 5399808 |
| CQ2 | 540 | 624 | 683 | 569664 | 623040 | 656000 |
| CQ3 | 72032 | 76200 | 80912 | 5399808 | 5399808 | 5399808 |
| CQ4 | 24418 | 44328 | 214008 | 2888064 | 2943616 | 2968448 |
| CQ5 | 18927 | 32700 | 62442 | 4259584 | 5399808 | 5399808 |
| CQ6 | 1603200 | 1925568 | 1925568 | 793184 | 935776 | 963552 |
| CQ7 | 13 | 17 | 26 | 825920 | 842784 | 883616 |
| CQ8 | 2534 | 2744 | 3106 | - | - | - |
| CQ9 | 73440 | 73440 | 73716 | 5399808 | 5399808 | 5399808 |
| CQ10 | 42094 | 52758 | 56458 | 5279744 | 5399808 | 5399808 |
| CQ11 | 2459 | 2673 | 2911 | 5399808 | 5399808 | 5399808 |
| CQ12 | 13544 | 16067 | 18589 | 3511040 | 3690624 | 3740672 |
| CQ13 | 1979 | 2481 | 3222 | 150 | 150 | 150 |
| CQ14 | - | - | - | - | - | - |
| SQ1 | 0 | 0 | 1 | 267136 | 292016 | 309088 |
| SQ2 | 179 | 326 | 666 | 4219392 | 4612608 | 5073152 |
| SQ3 | 1 | 2 | 3 | 589344 | 634304 | 1122176 |
| SQ4 | 0 | 0 | 0 | 3 | 5 | 13 |
| SQ5 | 0 | 0 | 0 | 280512 | 303488 | 322608 |
| SQ6 | 1 | 1 | 2 | 1350784 | 1418432 | 2006976 |
| SQ7 | 0 | 0 | 2 | 452848 | 562208 | 1772352 |
| UQ1 | - | - | - | - | - | - |
| UQ2 | 5 | 5 | 6 | 768544 | 904512 | 977024 |
| UQ3 | 4 | 4 | 5 | 713728 | 810624 | 938624 |
| UQ4 | 8 | 60 | 65 | 835008 | 2022592 | 2022592 |
| UQ5 | 3 | 4 | 5 | 816032 | 888544 | 1007904 |
| UQ6 | 582 | 749 | 786 | 2195072 | 2487552 | 2902400 |
| UQ7 | 577 | 729 | 798 | 2628736 | 2925312 | 3870080 |
| UQ8 | 3 | 3 | 4 | 465808 | 466752 | 466752 |

Table A.4.: All SF 30-100 Results for **RATG**.

| | LDBC-SNB SF1 | | | LDBC-SNB SF3 | | | LDBC-SNB SF10 | | |
|---|---|---|---|---|---|---|---|---|---|
| Query | 25%-Quantile | Median | 75%-Quantile | 25%-Quantile | Median | 75%-Quantile | 25Quantile–index7 | Median–index8 | 75Quantile–index9 |
| CQ1 | 253 | 289 | 323 | 625 | 653 | 708 | 1065 | 1118 | 1163 |
| CQ2 | 53 | 59 | 65 | 84 | 87 | 90 | 101 | 107 | 115 |
| CQ3 | 7068 | 8696 | 10143 | 17576 | 22551 | 25500 | 7762 | 18501 | 19565 |
| CQ4 | 84 | 213 | 356 | 1495 | 2953 | 5789 | 5278 | 10900 | 17361 |
| CQ5 | 1541 | 2488 | 4263 | 3072 | 5126 | 9661 | 1489 | 1732 | 2352 |
| CQ6 | 2703 | 2953 | 3250 | 11714 | 13260 | 14637 | 72392 | 91752 | 98864 |
| CQ7 | 22 | 36 | 52 | 2 | 2 | 3 | 1 | 1 | 2 |
| CQ8 | 2 | 2 | 3 | 33 | 43 | 53 | 1 | 1 | 2 |
| CQ9 | 5524 | 5917 | 6647 | 13949 | 15155 | 16665 | 30912 | 31087 | 34824 |
| CQ10 | 249 | 281 | 338 | 535 | 603 | 699 | 943 | 1104 | 1358 |
| CQ11 | 20 | 26 | 31 | 36 | 45 | 57 | 54 | 71 | 85 |
| CQ12 | 574 | 695 | 825 | 914 | 1025 | 1217 | 1217 | 1322 | 1397 |
| CQ13 | 2 | 3 | 4 | 2 | 4 | 4 | 2 | 2 | 3 |
| CQ14 | - | - | - | - | - | - | - | - | - |
| SQ1 | 1 | 1 | 2 | 1 | 1 | 1 | 0 | 1 | 1 |
| SQ2 | 3 | 5 | 9 | 3 | 5 | 11 | 3 | 6 | 14 |
| SQ3 | 1 | 2 | 2 | 1 | 2 | 2 | 1 | 2 | 2 |
| SQ4 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| SQ5 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| SQ6 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| SQ7 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| UQ1 | 96 | 96 | 96 | - | - | - | - | - | - |
| UQ2 | 2 | 3 | 3 | 2 | 2 | 3 | 2 | 2 | 3 |
| UQ3 | 2 | 3 | 3 | 2 | 2 | 3 | 2 | 2 | 3 |
| UQ4 | 3 | 4 | 4 | 3 | 4 | 5 | 3 | 3 | 4 |
| UQ5 | 2 | 3 | 3 | 2 | 2 | 3 | 2 | 2 | 3 |
| UQ6 | 3 | 3 | 4 | 3 | 3 | 4 | 3 | 3 | 4 |
| UQ7 | 3 | 4 | 4 | 3 | 4 | 4 | 3 | 3 | 4 |
| UQ8 | 2 | 3 | 3 | 2 | 2 | 3 | 2 | 2 | 2 |

Table A.5.: All SF 1-10 Results for **Neo4j**.

| | LDBC-SNB SF30 | | |
|---|---|---|---|
| Query | 25%-Quantile | Median | 75%-Quantile |
| CQ1 | 1915 | 2024 | 2077 |
| CQ2 | 137 | 143 | 149 |
| CQ3 | 48976 | 55202 | 90392 |
| CQ4 | 35632 | 63828 | 104804 |
| CQ5 | 17108 | 26532 | 46608 |
| CQ6 | 528128 | 580544 | 597888 |
| CQ7 | 1 | 1 | 2 |
| CQ8 | 95 | 119 | 167 |
| CQ9 | 45620 | 58496 | 63142 |
| CQ10 | 1695 | 1930 | 2707 |
| CQ11 | 74 | 87 | 104 |
| CQ12 | 541 | 672 | 772 |
| CQ13 | 6 | 6 | 7 |
| CQ14 | - | - | - |
| SQ1 | 0 | 1 | 1 |
| SQ2 | 5 | 13 | 27 |
| SQ3 | 2 | 3 | 7 |
| SQ4 | 1 | 1 | 1 |
| SQ5 | 0 | 1 | 1 |
| SQ6 | 0 | 1 | 1 |
| SQ7 | 0 | 1 | 1 |
| UQ1 | 2 | 3 | 3 |
| UQ2 | 2 | 3 | 3 |
| UQ3 | 3 | 11 | 12 |
| UQ4 | 2 | 2 | 3 |
| UQ5 | 3 | 3 | 4 |
| UQ6 | 12 | 12 | 13 |
| UQ7 | 2 | 2 | 3 |
| UQ8 | | | |

Table A.6.: All SF 30-100 Results for **Neo4j**.

# Bibliography


[Ada03] Yoshinobu Adachi. “Overview of partial model query language.” In: *ISPE CE*. 2003, pp. 549–555 (cit. on p. 160).

[Ada10] Christopher Adamson. *Star schema the complete reference*. McGraw Hill Professional, 2010. Chap. 1 (cit. on p. 160).

[AG08] Renzo Angles and Claudio Gutiérrez. “Survey of graph database models”. In: *ACM Comput. Surv.* 40.1 (2008), 1:1–1:39. DOI: 10.1145/1322432.1322433. URL: https://doi.org/10.1145/1322432.1322433 (cit. on p. 7).

[Ang+20] Renzo Angles et al. “The LDBC Social Network Benchmark”. In: *CoRR* abs/2001.02299 (2020). arXiv: 2001.02299. URL: http://arxiv.org/abs/2001.02299 (cit. on p. 15).

[Ang18] Renzo Angles. “The Property Graph Database Model”. In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*. Ed. by Dan Olteanu and Barbara Poblete. Vol. 2100. CEUR Workshop Proceedings. CEUR-WS.org, 2018. URL: http://ceur-ws.org/Vol-2100/paper26.pdf (cit. on p. 17).

[Apa21] Apache Software Foundation. *AGE - A Graph Extension For PostgreSQL*. 2021. URL: https://age.incubator.apache.org/# (visited on 07/11/2021) (cit. on p. 12).

[Ast+76] Morton M. Astrahan et al. “System R: Relational Approach to Database Management”. In: *ACM Trans. Database Syst.* 1.2 (1976), pp. 97–137. DOI: 10.1145/320455.320457. URL: https://doi.org/10.1145/320455.320457 (cit. on p. 120).

[Bay18] Matthias Bay. “Vergleich von PostGIS und GeoJSON zur Speicherung von Gebäudegeometrieinformationen”. Bachelor’s Thesis. Universtität Passau, 2018 (cit. on pp. 178, 180).

[Bee+10a] J. Beetz et al. “Towards an open building information model server: report on the progress of an open IFC framework”. English. In: *Proceedings of the 10th International Conference On Design and Decision Support Systems in Architecture and Urban Planning*. Eindhoven University of Technology, 2010 (cit. on p. 159).

[Bee+10b] Jakob Beetz et al. “BIMserver. org–An open source IFC model server”. In: *Proceedings of the CIP W78 conference*. 2010, p. 8 (cit. on pp. 159, 160, 170, 185, 202).

[Ben10] Jonathan Bennett. *OpenStreetMap*. Packt Publishing Ltd, 2010 (cit. on p. 205).

[Ber19] Florian Bergmüller. "Importing huge graphs into SQLGraph in OracleDB". Bachelor Thesis. University of Passau, 2019 (cit. on p. 52).

[BL01] Edward Barkmeyer and Joshua Lubell. "XML representation of EXPRESS models and data". In: *ICSE Workshop on XML Technologies and Software Engineering*. 2001 (cit. on p. 153).

[Bon13] Peter A. Boncz. "LDBC: benchmarks for graph and RDF data management". In: *17th International Database Engineering & Applications Symposium, IDEAS '13, Barcelona, Spain - October 09 - 11, 2013*. 2013, pp. 1–2. DOI: 10.1145/2513591.2527070. URL: https://doi.org/10.1145/2513591.2527070 (cit. on pp. 26, 123).

[Bor+13] Mihaela A. Bornea et al. "Building an efficient RDF store over a relational database". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. Ed. by Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias. ACM, 2013, pp. 121–132. DOI: 10.1145/2463676.2463718. URL: https://doi.org/10.1145/2463676.2463718 (cit. on pp. 11, 23, 37, 38, 40, 130).

[Bor+18] André Borrmann et al. "Building information modeling: Why? what? how?" In: *Building information modeling*. Springer, 2018, pp. 1–24 (cit. on p. 151).

[Bré79] Daniel Brélaz. "New Methods to Color Vertices of a Graph". In: *Commun. ACM* 22.4 (1979), pp. 251–256. DOI: 10.1145/359094.359101. URL: https://doi.org/10.1145/359094.359101 (cit. on p. 40).

[BVS18] Christian Bizer, Maria-Esther Vidal, and Hala Skaf-Molli. "Linked Open Data". In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York, NY: Springer New York, 2018, pp. 2096–2101. ISBN: 978-1-4614-8265-9. DOI: 10.1007/978-1-4614-8265-9_80603. URL: https://doi.org/10.1007/978-1-4614-8265-9_80603 (cit. on p. 203).

[Car+97] Michael J. Carey et al. "The BUCKY Object-Relational Benchmark (Experience Paper)". In: *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. Ed. by Joan Peckham. ACM Press, 1997, pp. 135–146. DOI: 10.1145/253260.253283. URL: https://doi.org/10.1145/253260.253283 (cit. on p. 159).

[CB74] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language". In: *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, USA, May 1-3, 1974, 2 Volumes*. Ed. by Randall Rustin. ACM, 1974, pp. 249–264. DOI: 10.1145/800296.811515. URL: https://doi.org/10.1145/800296.811515 (cit. on p. 13).

[Che13] Ruiwen Chen. "Managing massive graphs in relational DBMS". In: *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*. Ed. by Xiaohua Hu et al. IEEE Computer Society, 2013, pp. 1–8. DOI: 10.1109/BigData.2013.6691776. URL: https://doi.org/10.1109/BigData.2013.6691776 (cit. on p. 11).

[Com79] Douglas Comer. "The Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (1979), pp. 121–137. DOI: 10.1145/356770.356776. URL: https://doi.org/10.1145/356770.356776 (cit. on p. 62).

[Cou20] Linked Data Benchmark Council, ed. *The LDBC Social Network Benchmark (version 0.3.2)*. 2020. URL: http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf (cit. on pp. 69, 136).

[Dom+10] David Dominguez-Sal et al. "A Discussion on the Design of Graph Database Benchmarks". In: *Performance Evaluation, Measurement and Characterization of Complex Systems - Second TPC Technology Conference, TPCTC 2010, Singapore, September 13-17, 2010. Revised Selected Papers*. Ed. by Raghunath Othayoth Nambiar and Meikel Poess. Vol. 6417. Lecture Notes in Computer Science. Springer, 2010, pp. 25–40. DOI: 10.1007/978-3-642-18206-8_3. URL: https://doi.org/10.1007/978-3-642-18206-8%5C_3 (cit. on p. 125).

[Dom20] Michael Klaus Nguetsa Domo. "Using prefetching strategies to increase performance of algorithms in graph databases". MA thesis. University of Passau, 2020 (cit. on p. 209).

[Dur+18] Gabriel Campero Durand et al. "Piecing Together Large Puzzles, Efficiently: Towards Scalable Loading Into Graph Database Systems". In: *Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, Wuppertal, Germany, May 22-25, 2018*. Ed. by Gerhard Klassen and Stefan Conrad. Vol. 2126. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 95–100. URL: http://ceur-ws.org/Vol-2126/paper15.pdf (cit. on pp. 13, 46).

[Ehl15] Christoph Ehlers. "Top-k Semantic Caching". PhD thesis. Universität Passau, 2015, p. 266 (cit. on p. 71).

[Erl+15] Orri Erling et al. "The LDBC Social Network Benchmark: Interactive Workload". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 619–630. DOI: 10.1145/2723372.2742786. URL: https://doi.org/10.1145/2723372.2742786 (cit. on pp. 14, 123, 134).

[Fac16] Facebook Inc. *GraphQL*. 2016. URL: http://facebook.github.io/graphql (visited on 07/08/2021) (cit. on p. 13).

[Fra+18a] Nadime Francis et al. “Cypher: An Evolving Query Language for Property Graphs”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657. URL: https://doi.org/10.1145/3183713.3190657 (cit. on pp. 6, 14, 93).

[Fra+18b] Nadime Francis et al. “Formal Semantics of the Language Cypher”. In: *CoRR* abs/1802.09984 (2018). arXiv: 1802.09984. URL: http://arxiv.org/abs/1802.09984 (cit. on pp. 18, 73, 74, 76, 83–87, 90, 95).

[FS17] Burkhard Freitag and Alexander Stenzer. “MonArch – A Digital Archive for Cultural Heritage”. In: *Das Digitale und die Denkmalpflege: Bestandserfassung - Denkmalvermittlung - Datenarchivierung - Rekonstruktion verlorener Objekte*. Ed. by Gerhard Vinken and Birgit Franz. 2017. ISBN: 978-3-95954-030-8. URL: http://books.ub.uni-heidelberg.de/arthistoricum/catalog/book/263 (cit. on pp. 3, 17, 149).

[Goj21] Sabir Gojayev. “Efficiently executing data graph queries in SQL”. MA thesis. University of Passau, 2021 (cit. on pp. 71, 120, 121, 209).

[Gre+18] Alastair Green et al. “openCypher: New Directions in Property Graph Querying”. In: *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. Ed. by Michael H. Böhlen et al. OpenProceedings.org, 2018, pp. 520–523. DOI: 10.5441/002/edbt.2018.62. URL: https://doi.org/10.5441/002/edbt.2018.62 (cit. on pp. 14, 69).

[GS02] Rosalba Giugno and Dennis E. Shasha. “GraphGrep: A Fast and Universal Method for Querying Graphs”. In: *16th International Conference on Pattern Recognition, ICPR 2002, Quebec, Canada, August 11-15, 2002*. IEEE Computer Society, 2002, pp. 112–115. DOI: 10.1109/ICPR.2002.1048250. URL: https://doi.org/10.1109/ICPR.2002.1048250 (cit. on p. 13).

[Har20] Natasha Hartanto. “Benchmarking IFC Stores with the LDBC SNB Framework”. Bachelor’s Thesis. Universtität Passau, 2020 (cit. on p. 186).

[Her94] Axel Herbst. “Long-Term Database Support for EXPRESS Data”. In: *Seventh International Working Conference on Scientific and Statistical Database Management, September 28-30, 1994, Charlottesville, Virginia, USA, Proceedings*. Ed. by James C. French and Hans Hinterberger. IEEE Computer Society, 1994, pp. 207–216. DOI: 10.1109/SSDM.1994.336946. URL: https://doi.org/10.1109/SSDM.1994.336946 (cit. on pp. 153, 154).

[Hil20] Ben Hillinger. “Ein Konzept zur Speicherung von RDF-Graphen als Property Graph”. Bachelor Thesis. University of Passau, 2020 (cit. on pp. 48, 49, 59).

[HP13] Florian Holzschuher and René Peinl. "Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j". In: *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*. Ed. by Giovanna Guerrini. ACM, 2013, pp. 195–204. DOI: 10.1145/2457317.2457351. URL: https://doi.org/10.1145/2457317.2457351 (cit. on p. 13).

[HP17] Olaf Hartig and Jorge Pérez. "An Initial Analysis of Facebook's GraphQL Language". In: *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*. Ed. by Juan L. Reutter and Divesh Srivastava. Vol. 1912. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1912/paper11.pdf (cit. on p. 14).

[HP18] Olaf Hartig and Jorge Pérez. "Semantics and Complexity of GraphQL". In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. Ed. by Pierre-Antoine Champin et al. ACM, 2018, pp. 1155–1164. DOI: 10.1145/3178876.3186014. URL: https://doi.org/10.1145/3178876.3186014 (cit. on p. 13).

[HS08] Huahai He and Ambuj K. Singh. "Graphs-at-a-time: query language and access methods for graph databases". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. Ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 405–418. DOI: 10.1145/1376616.1376660. URL: https://doi.org/10.1145/1376616.1376660 (cit. on p. 13).

[INS17] Ali Ismail, Ahmed Nahar, and Raimar Scherer. "Application of graph databases and graph theory concepts for advanced analysing of BIM models based on IFC standard". In: *Proceedings of EGICE* (2017) (cit. on p. 161).

[ISO04] ISO/TC 184/SC 4 Industrial data. *Industrial automation systems andintegration - Product data representation and exchange- Part 11: Description methods: The EXPRESS language reference manual*. Standard. Geneva, CH: International Organization for Standardization, 2004 (cit. on pp. 153, 155).

[ISO18] ISO/TC 59/SC 13 Organization and digitization of information about buildings and civil engineering works, including building information modelling (BIM). *Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries — Part 1: Data schema*. Standard. Geneva, CH: International Organization for Standardization, 2018 (cit. on pp. 152, 154, 159).

[ISO21] ISO/TC 184/SC 4 Industrial data. *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles*. Standard. Geneva, CH: International Organization for Standardization, 2021 (cit. on pp. 155, 199).

[ISS18] Ali Ismail, Barbara Strug, and Grazyna Slusarczyk. “Building Knowledge Extraction from BIM/IFC Data for Analysis in Graph Databases”. In: *Artificial Intelligence and Soft Computing - 17th International Conference, ICAISC 2018, Zakopane, Poland, June 3-7, 2018, Proceedings, Part II*. Ed. by Leszek Rutkowski et al. Vol. 10842. Lecture Notes in Computer Science. Springer, 2018, pp. 652–664. DOI: 10.1007/978-3-319-91262-2_57. URL: https://doi.org/10.1007/978-3-319-91262-2%5C_57 (cit. on p. 161).

[Kem15] Alfons Kemper. *Datenbanksysteme - Eine Einführung, 10. Auflage*. De Gruyter Studium. de Gruyter Oldenbourg, 2015. ISBN: 978-3-11-044375-2. URL: https://www.degruyter.com/view/product/462324?rskey=RKIg0w%5C&result=1 (cit. on p. 120).

[KHJ98] David Koonce, Lizhong Huang, and Robert Judd. “EQL an Express Query Language”. In: *Computers and Industrial Engineering* 35.1 (1998), pp. 271–274. ISSN: 0360-8352. DOI: https://doi.org/10.1016/S0360-8352(98)00050-3. URL: https://www.sciencedirect.com/science/article/pii/S0360835298000503 (cit. on p. 160).

[Kid20] Omar Kidar. “Incremental Indexing of Very Large Graphs in Relational Databases”. MA thesis. University of Passau, 2020 (cit. on p. 66).

[Knu73] Donald E. Knuth. “The Art of Computer Programming, Volume III: Sorting and Searching”. In: (1973) (cit. on p. 49).

[Knu98] Donald E. Knuth. *The Art of Computer Programming - Volume 3: Sorting and Searching*. Second Edition. Boston: Addison-Wesley Professional, 1998. Chap. Section 5.4: External Sorting, 254–ff. ISBN: 0-201-89685-0 (cit. on p. 53).

[Kor16] Lana Sophia Kornev. “Anfragen relationaler Algebra in SQL”. Bachelor Thesis. University of Passau, 2016 (cit. on p. 71).

[Kor17] Lana Sophia Kornev. “Cypher für SQLGraph”. MA thesis. University of Passau, 2017 (cit. on pp. 22, 71, 123, 130).

[Kri] Thomas Krijnen. URL: http://ifcopenshell.org/ (cit. on p. 202).

[Kus13] Ruth Kusterer. *jMonkeyEngine 3.0 Beginner’s Guide*. Packt Publishing Ltd, 2013 (cit. on pp. 202, 204).

[Lee+14] Ghang Lee et al. “Query Performance of the IFC Model Server Using an Object-Relational Database Approach and a Traditional Relational Database Approach”. In: *Journal of Computing in Civil Engineering* 28 (Mar. 2014), pp. 210–222. DOI: 10.1061/(ASCE)CP.1943-5487.0000256 (cit. on pp. 159, 160).

[LMS05] Paul J. Leach, Michael Mealling, and Rich Salz. “A Universally Unique IDentifier (UUID) URN Namespace”. In: *RFC* 4122 (2005), pp. 1–32. DOI: 10.17487/RFC4122. URL: https://doi.org/10.17487/RFC4122 (cit. on p. 199).

[Maa09] Abdelsalam M. Maatuk. “Migrating relational databases into object-based and XML databases”. PhD thesis. Northumbria University, Newcastle upon Tyne, UK, 2009. URL: http://nrl.northumbria.ac.uk/3374/ (cit. on p. 160).

[Mar15] Angel Marquez. *PostGIS essentials*. Packt Publishing Ltd, 2015 (cit. on p. 178).

[MB13] Wiet Mazairac and Jakob Beetz. “BIMQL - An open query language for building information models”. In: *Adv. Eng. Informatics* 27.4 (2013), pp. 444–456. DOI: 10.1016/j.aei.2013.06.001. URL: https://doi.org/10.1016/j.aei.2013.06.001 (cit. on pp. 159, 160).

[MSV17] József Marton, Gábor Szárnyas, and Dániel Varró. “Formalising openCypher Graph Queries in Relational Algebra”. In: *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings*. Ed. by Marite Kirikova, Kjetil Nørvåg, and George A. Papadopoulos. Vol. 10509. Lecture Notes in Computer Science. Springer, 2017, pp. 182–196. DOI: 10.1007/978-3-319-66917-5_13. URL: https://doi.org/10.1007/978-3-319-66917-5%5C_13 (cit. on pp. 71, 77).

[Müh+13] Tobias Mühlbauer et al. “Instant Loading for Main Memory Databases”. In: *Proc. VLDB Endow.* 6.14 (2013), pp. 1702–1713. DOI: 10.14778/2556549.2556555. URL: http://www.vldb.org/pvldb/vol6/p1702-muehlbauer.pdf (cit. on p. 12).

[Mul11] Tony Mullen. *Mastering blender*. John Wiley & Sons, 2011 (cit. on p. 202).

[Nad+16] Irakli Nadareishvili et al. *Microservice architecture: aligning principles, practices, and culture*. " O’Reilly Media, Inc.", 2016 (cit. on p. 200).

[OH12] Regina Obe and Leo Hsu. *PostgreSQL - Up and Running: a Practical Guide to the Advanced Open Source Database*. O’Reilly, 2012. ISBN: 978-1-449-32633-3. URL: http://www.oreilly.de/catalog/9781449326333/index.html (cit. on p. 176).

[ÖV20] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020. Chap. 3.3. ISBN: 978-3-030-26252-5. DOI: 10.1007/978-3-030-26253-2. URL: https://doi.org/10.1007/978-3-030-26253-2 (cit. on p. 175).

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. “Semantics and complexity of SPARQL”. In: vol. 34. 3. 2009, 16:1–16:45. DOI: 10.1145/1567274.1567278. URL: https://doi.org/10.1145/1567274.1567278 (cit. on p. 13).

[PF00] Meikel Pöss and Chris Floyd. “New TPC Benchmarks for Decision Support and Web Commerce”. In: *SIGMOD Rec.* 29.4 (2000), pp. 64–71. DOI: 10.1145/369275.369291. URL: https://doi.org/10.1145/369275.369291 (cit. on p. 14).

[Rod15] Marko A. Rodriguez. “The Gremlin Graph Traversal Machine and Language (Invited Talk)”. In: *Proceedings of the 15th Symposium on Database Programming Languages*. DBPL 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 1–10. ISBN: 9781450339025. DOI: 10.1145/2815072.2815073. URL: https://doi.org/10.1145/2815072.2815073 (cit. on pp. 12, 69).

[RR08] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008 (cit. on p. 200).

[RWE15] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.", 2015. Chap. 6 (cit. on p. 142).

[Sca+18] Lucas C. Scabora et al. “Cutting-edge Relational Graph Data Management with Edge-k: From One to Multiple Edges in the Same Row”. In: *J. Inf. Data Manag.* 9.1 (2018), pp. 20–35. URL: https://periodicos.ufmg.br/index.php/jidm/article/view/413 (cit. on p. 11).

[Sch19a] Matthias Schmid. “An Approach to Efficiently Storing Property Graphs in Relational Databases”. In: *Proceedings of the 31st GI-Workshop Grundlagen von Datenbanken, Saarburg, Germany, June 11-14, 2019*. Ed. by Ralf Schenkel. Vol. 2367. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 56–61. URL: http://ceur-ws.org/Vol-2367/paper%5C_8.pdf (cit. on p. 12).

[Sch19b] Matthias Schmid. “On efficiently storing huge property graphs in relational database management systems”. In: *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services, iiWAS 2019, Munich, Germany, December 2-4, 2019*. ACM, 2019, pp. 344–352. DOI: 10.1145/3366030.3366046. URL: https://doi.org/10.1145/3366030.3366046 (cit. on p. 45).

[SEL17] Wawan Solihin, Charles Eastman, and Yong-Cheol Lee. “Multiple representation approach to achieve high-performance spatial queries of 3D BIM data using a relational database”. In: *Automation in Construction* 81 (2017), pp. 369–388 (cit. on p. 160).

[SES19] Alexander Stenzer, Christina Ehrlinger, and Matthias Schmid. “Ansätze zur semantischen 3D-Repräsentation von Bauwerken in Datenbanken”. In: *Der Modelle Tugend 2.0: Digitale 3D-Rekonstruktion als virtueller Raum der architekturhistorischen Forschung*. Ed. by Piotr Kuroczyński, Mieke Pfarr-Harfst, and Sander Münster. Vol. 2. Computing in Art and Architecture. Jan. 1, 2019, pp. 371–390. ISBN: 978-3-947449-70-5. DOI: https://doi.org/10.11588/arthistoricum.515.c7581. URL: https://books.ub.uni-heidelberg.de/arthistoricum/catalog/book/515/c7581. published (cit. on p. 198).

[Sha21] Anand Kumar Shanmugam. “Efficiently Executing Data Graph Queries in Oracle SQL”. Master Thesis. University of Passau, 2021 (cit. on pp. 29, 71, 145, 209).

[Shk] Shkale-Msft. *Graph processing with SQL Server and Azure SQL Database - SQL Server*. URL: https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-2017 (cit. on p. 11).

[SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. “Heuristic and Randomized Optimization for the Join Ordering Problem”. In: *VLDB J.* 6.3 (1997), pp. 191–208. DOI: 10.1007/s007780050040. URL: https://doi.org/10.1007/s007780050040 (cit. on p. 121).

[Sol+17] Wawan Solihin et al. “A simplified relational database schema for transformation of BIM data into a query-efficient and spatially enabled database”. In: *Automation in Construction* 84 (2017), pp. 367–383 (cit. on p. 160).

[Ste+08] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008 (cit. on p. 202).

[Ste18] Alexander Stenzer. “Ein Ansatz zur semantik-basierten Anfragerelaxation für hierarchische Strukturen”. PhD thesis. Universität Passau, Jan. 1, 2018. URL: https://opus4.kobv.de/opus4-uni-passau/files/574/Alexander_Stenzer+_Anfrage-Relaxation.pdf. published (cit. on p. 150).

[Sun+15] Wen Sun et al. “SQLGraph: An Efficient Relational-Based Property Graph Store”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1887–1901. DOI: 10.1145/2723372.2723732. URL: https://doi.org/10.1145/2723372.2723732 (cit. on pp. 6, 11, 21–24, 36, 124, 131, 139, 145).

[SW94] Douglas A. Schenck and Peter R. Wilson. *Information Modeling: The EXPRESS Way*. USA: Oxford University Press, Inc., 1994. ISBN: 0195087143 (cit. on p. 153).

[Szá+18] Gábor Szárnyas et al. “An early look at the LDBC social network benchmark’s business intelligence workload”. In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018*. Ed. by Akhil Arora et al. ACM, 2018, 9:1–9:11. DOI: 10.1145/3210259.3210268. URL: https://doi.org/10.1145/3210259.3210268 (cit. on pp. 15, 123).

[TAS06] Niraj Tolia, David G. Andersen, and Mahadev Satyanarayanan. “Quantifying Interactive User Experience on Thin Clients”. In: *Computer* 39.3 (2006), pp. 46–52. DOI: 10.1109/MC.2006.101. URL: https://doi.org/10.1109/MC.2006.101 (cit. on pp. 195, 196).

[TBS16] Eike Tauscher, Hans-Joachim Bargstädt, and Kay Smarsly. “Generic BIM queries based on the IFC object model using graph theory”. In: *Proceedings of the 16th International Conference on Computing in Civil and Building Engineering, Osaka, Japan*. 2016, pp. 6–8 (cit. on p. 160).

[Tha+19] Harsh Thakkar et al. “Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin (Extended Version)”. In: vol. abs/1908.06265. 2019. arXiv: 1908.06265. URL: http://arxiv.org/abs/1908.06265 (cit. on p. 13).

[The+16] Manuel Then et al. “Evaluation of parallel graph loading techniques”. In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*. Ed. by Peter A. Boncz and Josep Lluís Larriba-Pey. ACM, 2016, p. 4. DOI: 10.1145/2960414.2960418. URL: https://doi.org/10.1145/2960414.2960418 (cit. on pp. 12, 13, 46).

[The11] Volker Thein. “Industry foundation classes (IFC)”. In: *BIM interoperability through a vendor-independent file format* (2011) (cit. on p. 152).

[Vog19] Andreas Vogt. “Ein View-Konzept für Graphdatenbanken am Beispiel IFC”. MA thesis. University of Passau, 2019 (cit. on p. 209).

[Wal15] Craig Walls. *Spring Boot in action*. Simon and Schuster, 2015 (cit. on p. 202).

[YL21] Huaquan Ying and Sanghoon Lee. “A rule-based system to automatically validate IFC second-level space boundaries for building energy analysis”. In: *Automation in Construction* 127 (2021), p. 103724. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2021.103724. URL: https://www.sciencedirect.com/science/article/pii/S0926580521001758 (cit. on p. 163).

[Zha93] Yanchun Zhang. “On Horizontal Fragmentation of Distributed Database Design”. In: *Advances in Database Research - Proceedings of the 4th Australian Database Conference, ADC ’93, Griffith University, Brisbane, Queensland, Australia, February 1-2, 1993*. Ed. by Maria E. Orlowska and Mike P. Papazoglou. World Scientific, 1993, pp. 121–130 (cit. on p. 175).

# List of Figures

# List of Definitions

# List of Tables

# Listings

# List of Algorithms