TU WIEN Informatics

Analyse und Vergleich von gängigen Graph-Anfragesprachen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martin Klampfer, BSc
Matrikelnummer 01526110

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Univ.Ass. Dipl.-Ing. Matthias Lanzinger, BSc

Wien, 9. Dezember 2020

Martin Klampfer | Reinhard Pichler

TU WIEN Informatics

Analysis and Comparison of Common Graph Query Languages

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Martin Klampfer, BSc
Registration Number 01526110

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Assistance: Univ.Ass. Dipl.-Ing. Matthias Lanzinger, BSc

Vienna, 9th December, 2020

Martin Klampfer | Reinhard Pichler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Martin Klampfer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. Dezember 2020

Martin Klampfer

Danksagung

Ich möchte mich ganz herzlich bei meinem Betreuer, Prof. Dr. Reinhard Pichler, für seine Unterstützung durch den ganzen Diplomarbeits-Prozess bedanken. Er unterstützte mich bei der Auswahl eines spannenden Themas, ließ mir nützliche Unterlagen zukommen und war immer für einen Ratschlag erreichbar. Außerdem gilt mein herzlicher Dank meinen Eltern und meinem Bruder, die mich während meiner Studienzeit immer unterstützten, ermutigten und anspornten.

Acknowledgements

First and foremost I want to thank my supervisor Prof. Dr. Reinhard Pichler for his support and guidance throughout the whole process. He not only helped me when first choosing a topic but provided valuable materials and was always reachable when I needed some advice. I would also like to thank my parents and my brother, they not only supported me with my wish to study but also encouraged me throughout those years.

Kurzfassung

Graphdatenbanken werden immer öfter verwendet um vernetzte Daten abzuspeichern. Viele dieser Systeme bieten ein flexibles Datenmodell basierend auf *property graphs*, das sind Graphen wo Knoten und Kanten mit Tags gekennzeichnet und weitere Eigenschaften in Schlüssel-Wert Paaren gespeichert werden können. Um auf Daten in solch einer Datenbank zuzugreifen verwendet man Anfragesprachen. Im Unterschied zu relationalen Datenbanken, wo SQL die standardisierte Anfragesprache ist, gibt es noch keine standardisierte Anfragesprache für Graphdatenbanken. Nutzer können daher von einer breiten Palette an Sprachen wählen die sich in dem verwendeten Datenmodell, der Ausdrucksstärke, der Einfachheit der Verwendung und so weiter unterscheiden.

Das Ziel dieser Arbeit ist ein systematischer Vergleich von solchen Sprachen. Daraus wollen wir Hilfestellungen zur Sprachauswahl bei bestimmten Anwendungsfällen ableiten. In einem ersten Schritt identifizieren und analysieren wir Kernfunktionen die in allen Anfragesprachen für Graphdatenbanken vorhanden sind. Dazu zählt das Finden von Teilgraphen anhand von Graphmustern (patterns), von Pfaden mittels Pfadanfragen (path queries) und die Kombination dieser beiden zu Navigationsanfragen (navigational queries). Wir erweitern diese Funktionen um strukturunabhängige Anfragen sowie Datenmanipulations- und Datendefinitions- Operationen.

Anhand dieser Funktionen analysieren wir fünf moderne Anfragesprachen für Graphdatenbanken: Cypher, Gremlin, PGQL, GSQL und G-CORE. Wir analysieren die Sprachen nicht nur anhand von generellen Charakteristika, sondern implementieren auch Teile des “Social Network Benchmarks” und analysieren diese Anfragen. Es stellt sich heraus, dass Cypher, PGQL und G-CORE eine ähnliche Syntax verwenden sowie ähnlich ausdrucksstark und einfach zu verwenden sind. Auf der anderen Seite ähneln sich GSQL und Gremlin: beide sind Turing-vollständig und unterstützen nicht nur deklarative, sondern auch imperative Konstruktionen.

Abstract

Graph databases are increasingly adopted in industry as they offer native support for graph-like data. Many such systems use a flexible data model based on property graphs, these are graphs where vertices and edges can be labeled and augmented with key-value pairs, the properties. Data stored in such a database is accessed via graph query languages. Unlike SQL, that is the standardized query language for relational databases, there is no standard for a graph query language yet. Users can therefore choose from a range of languages that vary in their specific data model, expressiveness, ease of use and so on.

The goal of this thesis is a systematic comparison of graph query languages and to give guidelines for choosing a language for specific application scenarios. Towards achieving this goal, we identify and analyze core features inherent to such languages. The most common ones are subgraph discovery by matching graph patterns, path discovery using path queries and the combination of both in navigational queries. We extend these features to also include structure independent queries as well as data manipulation and data definition operations.

Based on these features, we then analyze five contemporary graph query languages: Cypher, Gremlin, PGQL, GSQL and G-CORE. The analysis is not only based on general characteristics of the languages as we also implement parts of the Social Network Benchmark and analyze these queries. It turns out that Cypher, PGQL and G-CORE are quite similar in their syntax, expressiveness and ease of use. On the other hand, GSQL and Gremlin share some similarities as they are both Turing-complete languages that are not solely declarative but also offer imperative constructs.

Contents

CHAPTER 1

Introduction

Relational databases are well studied and optimized for data that can be modeled via tabular relations. However, using a relational database comes with serious restrictions, especially when the information is highly interconnected. In this case, querying data often means to move along some relationships between the entities. This is typically done by joining multiple relations, which adds computational overhead compared to systems tailored for such queries. Furthermore, relational databases are not well suited to handle a dynamic model where the structure changes over time.

Graph databases are a type of a NoSQL database and specifically developed to handle graph-like data, where information coming from relationships is at least as important as information stored in entities [Ang12]. They are increasingly adopted in industry as for example chemistry-, biology-, or social network- related data can often be modeled as a graph with information stored in nodes and relationships connecting them. In these cases graph databases offer a more natural and flexible model as well as native support for queries on a graph structure, like navigating along the relationships [AG08, RWE15]. Graph databases have been around since the 1980s and enjoyed a rise in popularity over the last decade. This comes from technical advancements that allow large graphs like social networks with millions or even billions of nodes to be stored and processed [AG18].

Data stored in a graph database is accessed via a query language. Unlike for relational database management systems[1] , where SQL is the standardized query language, there is no standard for a graph query language yet, as work on such a standard started in 2019. Therefore, many vendors of graph database systems define and use their own query language that differs from others in their expressiveness, their supported data types, their ease of use and so on. The rise of interest in graph databases is accompanied by

[1] We use the terms database management system, database system or simply database interchangeably for the system that *"enables users to define, create, maintain and control access"* [CB05] to a collection of data.

academia putting more focus on this area as well. This resulted in a better understanding of some systems and their underlying structures, and also in the invention of more query languages.

Choosing a system therefore involves not only the properties of the system but also the supported query language. With many languages to choose from, it is important to get an overview about their expressiveness, their characteristics and their ease when formulating queries. But to our knowledge there is no comprehensive comparison of current major and most promising languages regarding these factors.

This thesis provides a comparison of common query languages, especially regarding their expressiveness and ease of use. To get a comprehensible comparison, core features of query languages are identified and analyzed, and the comparison is done based on these features. We analyze four languages from industry (Cypher, Gremlin, PGQL, GSQL) and one promising language from a research project (G-CORE). The analysis is not only based on general information as we also implement parts of the Social Network Benchmark. Whereas a full implementation of all chosen queries can be found on our GitHub repository[2], we analyze only some of them in detail in this thesis.

It turns out that Cypher is a good choice, both when first starting with graph databases and for the use in applications, as long as more expressive path queries and a schema are not needed. PGQL is very similar to Cypher, and what it lacks in functionality it makes up for with sophisticated path querying capabilities. Although Gremlin allows for expressive queries, its low-level approach with a focus on imperative traversals makes it harder to get into for someone familiar with high-level declarative query languages. GSQL is designed for huge graphs with support for native parallel query execution. That results in a slightly different syntax compared to Cypher and PGQL, but the major difference is the support of a strong type system and schema. We conclude with G-CORE, an expressive research language that combines useful features from existing languages, achieves full composability and elevates paths to first-class citizens in the graph.

The remainder of this thesis is structured as follows: Chapter 2 introduces graph databases in general, including their underlying data models and the query languages we focus on in our comparison. We then identify common features of query languages and analyze these languages regarding the expressiveness they enable in Chapter 3. The five query languages are introduced and analyzed in Chapter 4. We end this chapter with a general comparison and provide guidelines for selecting a language for specific application scenarios. Finally, Chapter 5 concludes the thesis and includes notes on future work.

[2] https://github.com/martin-kl/Diploma_Thesis_01526110_code

CHAPTER 2

Graph Databases

Since the 1980s, a multitude of database systems supporting different graph models and query languages targeting different applications have been introduced. We start with a description of graph databases in general and give some historical background in Section 2.1. The basic concepts of graph databases and underlying data models are introduced in Section 2.2 and Section 2.3, respectively. Section 2.4 gives an introduction to five of the most common graph query languages. These are the languages we analyze and compare in Chapter 4.

2.1 Brief History and Scope

Graph databases and query languages have been around for multiple decades. Initially introduced in the 1980s, with the invention of graph databases like G-BASE in 1987 [Kun87] and corresponding data models and query languages like G [CMW87], interest in the area declined in the later 1990s as databases targeting for example spatial and geographical data emerged [AG08]. In the late 2000s and especially during the last decade we saw a breakthrough of graph databases, as the need for frequent schema changes and real-time query responses on databases containing vast amounts of interconnected data became more relevant [FB18]. They are now used to handle data from a wide range of fields: from transport networks over biological information like genes and protein interactions to social networks [RWE15].

Using a graph database became a requirement to handle vast amounts of interconnected data in fields like these, as they are designed to allow a more natural and faster access to graph-like data [Ang12]. As an example for this, assume that we model such data in a relational database. This could be done by storing the edges in a table where each row represents an edge that contains two foreign key references, one for each adjacent node. We end up with lots of many-to-many relationships for all the connections carrying data. Although we are able to model graphs in this way, querying the structure can

become quite cumbersome. In contrast to conventional data management we are not only interested in the values stored in the tables but also in the connection patterns [EAL+15]. If we search for such patterns or query not only information stored in a single relation, we use join operations to move along a path [1] in the graph. These joins can easily become complex, therefore computationally demanding and impact the performance, even if only a small part of the graph is queried. Graph databases on the other hand are designed to store and process such interconnected data. They can handle large graphs while providing a high performance, especially for queries traversing them [HP16].

One can split the numerous available systems handling graph data into two main categories based on their focus on either transactional or analytical processing: [RWE15, RH19]

Transactional Systems that are focused on transactional processing are usually online, meaning that they are built for real-time access to the current data. Similar to the world of relational databases they are designed for *Online Transactional Processing (OLTP)* and called graph database management systems or simply graph databases. We focus on these tasks and systems in our thesis.

Analytical Systems focused on analyzing graph-like data can either be online or offline, i.e. they can work on the current data or on an offline copy of it. Regardless of this design choice, these tasks are often called *Online Analytical Processing (OLAP)*[2] and are typically done in batch-processing fashion. Graph databases support these tasks to some degree but are not optimized for them and sometimes even lack the ability to perform them at all. Therefore, special technologies exist that are often called graph compute engines or graph analytics systems.

2.2 Basic Concepts

Unlike relational databases that are built on tabular relations as underlying data structure, graph databases are built on the mathematical structures of graphs. Although there are multiple different data models used in various graph databases (see Section 2.3), they all represent their domain using its structure via nodes (vertices), edges (relationships, links) and in most cases properties. Unless mentioned otherwise, we assume directed graphs as most systems do not support undirected edges. Let's look at an example for such a data structure. Figure 2.1 depicts a small social network containing three nodes representing its users that are connected via multiple edges. Edges and nodes are either labeled as `User` or `FOLLOWS` and each node has a property called `name`. The structure tells us that Alice follows Bob, Carol follows Alice and Carol and Bob follow each other. This is a small motivating example of a social network that could easily be extended to also contain further information about the users or relationships by adding more properties.

[1] Moving along a path, that is a concatenation of adjacent edges, means to traverse the path from its start to its end node.

[2] They are sometimes also denoted as *Offline Analytical Processing* [BPG+19].

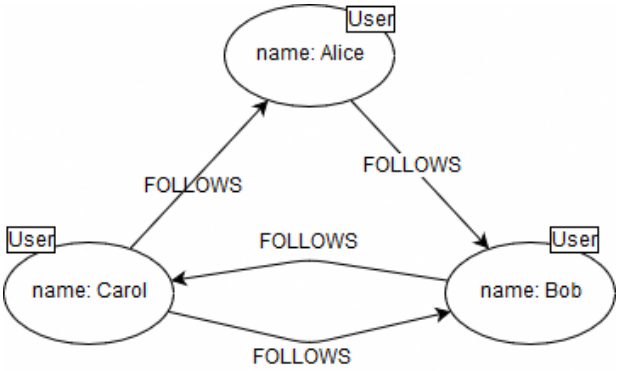


Figure 2.1: A small social graph.

Furthermore, the graph could also be extended to include other entities like posts or messages by adding appropriate nodes and edges.

Depicting information in such a connected data model allows a graph database to handle highly interconnected data well, but one has to keep in mind that building on graphs comes with its own drawbacks. While querying for related data and traversing the graph can be done efficiently, querying the global structure is often challenging and shows notably worse performance compared to relational databases in some cases. An example for such a global query would be to search for all users at a specific age or for persons that have at least one friend following them (i.e. one incoming edge labeled `FOLLOWS` in our example). As graph databases do not provide particularly efficient solutions for all problems, they are not seen as a replacement for their relational counterparts but more as a complementary system that is used especially to analyze strongly interrelated data. [FB18]

2.2.1 Native vs Non-Native Graph Representation

Databases with built-in support for graphs can be split into two categories that differ in their internal data representation: native (also pure) graph databases and multi-model (also non-native) ones [EAL+15, AG18, BPG+19].

- *Native graph databases* like Neo4j[3] and TigerGraph[4] focus solely on data structured in graphs. As their data model and query languages are specifically designed with graphs in mind, graphs are first class citizens in these systems. Most of these databases do not only present the data as a graph to the user but also use a graph data structure internally. This allows them to be optimized for certain graph queries. An example would be the provision of constant time adjacency retrievals regardless of the size of the graph that is possible in such systems. Furthermore,

[3] `https://neo4j.com/`
[4] `https://www.tigergraph.com/`

many such systems provide graph specific functions like the shortest path algorithm or breadth-first search out of the box.

- *Multi-model databases* support other models apart from graphs, like the relational or document oriented one, as well. Many of these systems transform the data into a different model to store it. Consequently, a global index is commonly used to store and retrieve adjacency information. Although some of these systems also provide graph specific functions out of the box, they are in many cases not that performant caused by the use of different storage and processing mechanisms. On the other hand, multi-model databases can integrate data coming in various different structures and possibly from multiple sources into a single system. Having all data in a single system allows for a uniform access via a single query language and the data can be analyzed over different structures. Examples for such systems are ArrangoDB[5] and Azure Cosmos DB[6].

Regardless of a graph database being categorized as native or multi-model we can identify two major components in such a system: the storage and the processing component [RWE15]. Although these components are common in both database categories, they differ substantially and their design and mechanisms influence the category of a database.

Storage The storage component is responsible for reliable storage and access to the data. One possibility to achieve this is to serialize the graph and store it in a relational table. This allows for the use of a single copy of the data that can be accessed by either SQL queries or queries written in a graph query language, depending for example on the knowledge of the engineers or the ease-of-use for the given problem. However, serializing the graph and keeping it in a data model that is not specifically designed and optimized for such interconnected data comes with all performance drawbacks mentioned earlier. This is one of the reasons why databases with such a *non-native* storage component are not categorized as native graph databases, but as multi-model ones.

Another possibility is to store data directly in a connected structure that allows for faster, in many cases constant time, retrievals of adjacent entities. By using such a storage component, these databases are usually able to scale notably better to large graphs containing millions of nodes while still providing a high performance for most graph related queries. A storage component built on such a design is referred to as *native graph storage*. As mentioned earlier, most native graph databases represent the data internally as a graph and use such a native graph storage.

Processing The processing component connects to the storage component and is responsible for all operations on the data. As the choice of the storage mechanism heavily influences the inner workings of the processing component, like the use of a

[5] https://www.arangodb.com/
[6] https://azure.microsoft.com/services/cosmos-db/

native graph storage that provides different performance characteristics and query possibilities to the processing component, the border between these components is quite blurry. Therefore, the most interesting part of the processing component concerns also the storage technique and is about the question whether the system relies on index-free adjacency or not.

When using *index-free adjacency*, adjacent nodes and edges are physically stored next to each other with direct references to each other. These components are also referred to as *native graph processing* and are used in most native graph databases. As an example, assume that we have the nodes x, y and the relationship $e = (x, y)$ in a system. With index-free adjacency, the processing component can "move" from x to e and further to y by directly retrieving their information without the need of a global index containing the relationships. Having these direct pointers allows for constant time retrieval of an adjacent node or relationship, regardless of the size of the whole graph. On the downside, graph databases in general and especially native graph processing systems that rely on their local adjacency information instead of global indexes have the problem that sharding a graph is very hard. Sharding is the process of partitioning a graph into multiple subgraphs that can be stored on different machines to achieve horizontal scalability. [RN11]

Another option for the design of a processing component is to use a global index that contains references to relationships and the nodes in these relationships. To navigate the graph, data is joined with the global index. This approach is used in multi-model graph databases.

2.2.2 Indexes

Indexes are used to speed up access to relevant data. Relational databases rely heavily on these indexes, as they enable tables to be efficiently joined and data entries to be efficiently found. Graph databases also rely on indexes in various ways. Systems not built on the principle of index-free adjacency use an index containing adjacency information. This index can then be used to move between connected entities in the graph by joining the data with the index. Depending on the system, this can work similar to relational databases. Graph databases relying on index-free adjacency have local indexes integrated into the graph structure. As briefly explained in the processing component above, every node and edge in such a system keeps direct pointers to all adjacent nodes and edges. These are stored together with the entity and form a local index for every node and edge. Once the start node of a traversal is found, they can traverse the neighborhood without accessing a dedicated or global index by using the pointers in their local index [BPG+19]. This is a significant advantage compared to systems without index-free adjacency where performance usually deteriorates as the graph becomes bigger, caused by an increase in the size of the index. However, these databases also use additional indexes to find specific nodes in the graph, like the start node in a query, and to accelerate other operations, especially global ones. As an example, assume that our social network from Figure 2.1 also contains the age of every user. If we want to get the names of all users older than 20

years for example, an index on the age allows the system to check for the condition on the index, and limits the access to nodes that satisfy it. [RN11]

2.3 Graph Data Models

Depending on the definition, the term *data model* (also database model, db-model or model) comprises not only the underlying data structure used by a database system but also the data description, integrity constraints, maintenance and even access and query mechanisms [AG08, Ang18]. We focus on the underlying data structure used by a database system in this section. As briefly mentioned earlier, it is defined around the mathematical structures of graphs, nodes and edges [Ang12]. Unlike in the world of relational databases there is no standard data model used in graph databases. Figure 2.1 is an example of a *labeled property graph* that is widely used, especially in native graph databases. Other notable models include the *Resource Description Framework (RDF)* model used in the domain of Semantic Web and *hypergraphs* that generalize the notion of a graph to allow edges between more than two nodes [AG18]. We now take a closer look at the RDF and various property graph models as these are the data models used in most graph databases of our time.

2.3.1 RDF

The RDF model encodes data in *subject-predicate-object* expressions and databases using this model are therefore also denoted as *Triplestore*. In contrast to other models, it uses a schema that is integrated together with the data into a single graph. The schema is a small part of the graph and contains information about possible connections as well as node and edge types. Since it is a World Wide Web Consortium (W3C) recommendation, it enjoys widespread recognition, good documentation and the available systems are unified by a common and standardized query language: SPARQL. It was originally developed to represent web metadata, but can also be used in a more general way to model interconnected resources using the aforementioned triples. On the other hand, these systems are usually not native graph databases as many transform the triples into other formats and store them for example in relational tables. Nonetheless, they fall under the category of graph databases as larger amounts of these triples can form a body of interconnected nodes and therefore a graph structure. [RWE15, AG08, AG18]

Figure 2.2 gives an example of an RDF graph containing the data from our social network example. An example of a triple in this case is “Alice follows Bob”, where `Alice` is the subject (the resource that is described), `follows` the predicate (property) and `Bob` the object (or property value) [AG08]. The schema is given in a separate part of the graph that is connected to the instances via `type`-edges. These connections represent the information given by the labels in Figure 2.1. Apart from the missing labels there are also no inherent properties on nodes. Such a property is stored in a triple that introduces a node containing the value and an edge labeled by the name of the property. An example for this is the node `"Alice"` connected by the `name`-edge to the node `user1`.

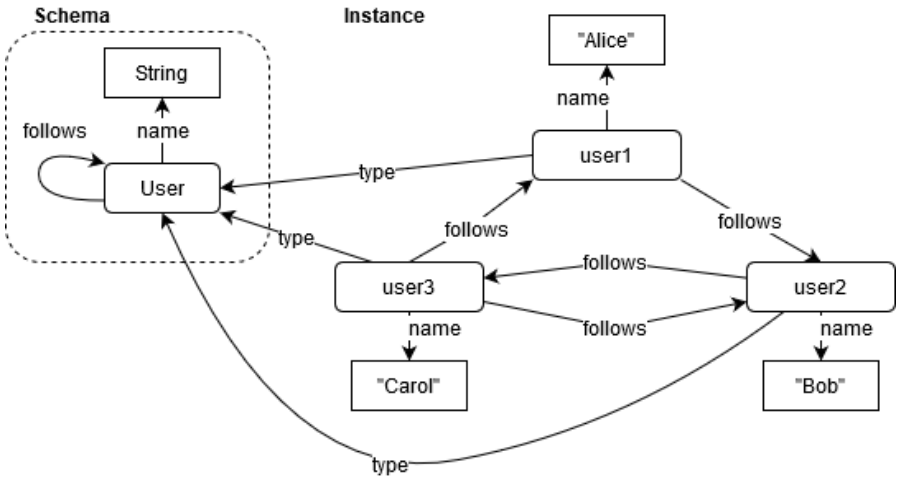


Figure 2.2: A small social graph in the RDF data model.

Despite being endorsed by W3C and having, in contrast to other graph databases, a standardized query language, these RDF stores are mainly used in the domain of Semantic Web. A reason for the lack of utilization in other domains could be the comparably bloated data structure that uses an additional node and edge for every label and property. Another reason could be the use of a schema that limits the flexibility of the data model [SS19].

2.3.2 Labeled (Property) Graphs

Edge-labeled graphs are one of the simplest form of graphs where nodes and edges can be each labeled with one label. Labels assigned to edges indicate the type of the relationship between the nodes. This allows us to model different relationships as well as multiple edges between any two nodes in a single graph. RDF-graphs are built on the basis of edge-labeled graphs but differ in their use of a schema and therefore the use of node and edge types. Edge-labeled graphs do not provide a schema and types can only be given implicitly by setting an appropriate label. Figure 2.3 shows the data from our small social network example encoded in an edge-labeled graph. We added the information that Carol is the sister of Alice to show the power of edge labels. Without these labels, we would simply have two edges from Carol to Alice without being able to distinguish them. [AAB+17]

(Labeled) Property graphs, also LPGs or simply property graphs, extend this model by adding labels to nodes and including properties directly in the nodes and edges. Properties, also called attributes, are key-value pairs containing non-graphical information like `name: Carol`. The inclusion of properties allows for a more natural data modeling and faster retrieval of the data related to a node or edge [Ang12]. Therefore, property graphs can be described as a "directed, labeled, attributed multigraph" [AG18], where multigraph denotes that multiple edges between any two nodes are possible. Although this model

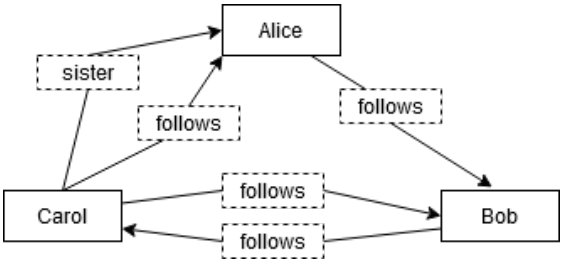


Figure 2.3: A small social graph in the edge-labeled data model.

is more popular than others [FGG+18b, BPG+19] and used in many graph databases, it is not standardized and many variants of it exist. Angles [Ang18] presents a formal definition of the property graph data model that allows nodes and edges to be labeled with multiple labels and every property to be assigned a set of values. Other definitions for example restrict edges to be labeled by only a single label [FGG+18b]. Some papers also require the use of a unique identifier for every node and edge, or restrict properties to only store single values instead of sets [AAB+17].

Let's look at an enriched example based on our small social network. The property graph in Figure 2.4 contains a unique identifier for every node and edge. This allows for fast and clear identification of entities and is also used in many graph databases. As property graphs do not have an explicit schema, the types of nodes and edges are given through labels attached to them. Our example allows a node to have multiple labels, like `User` and `Admin` on node `n1`. In contrast to the RDF model from Figure 2.2, the properties of a node are stored inside the node itself. This results in a more compact model as we do not need an additional node for every property. As mentioned before, edges can also have properties like the `since` attributes on the `follows` relationships.

Property graphs are very flexible as they do not use an explicit schema. One can simply omit properties that are not available for some parts in the graph, like the missing age attribute in the node `n3`. Likewise, other properties can be added to nodes, like the `mail` property in `n3`. Furthermore, if we want to add new information like a picture that is posted by one of our users, we can add a node labeled `Picture` and connect it via an edge labeled `posted` to the user that posted it.

Since most native graph databases use the property graph data model, they do not have a defined schema and are therefore more flexible compared to relational or other schema-dependent databases. We will focus on the property graph model and some variations of it in the rest of this thesis as many popular graph query languages are designed to operate on some variation of that model.

2.4 Query Languages

A query language is a high-level language used to access data stored in a database [ARV19]. In the strictest definition, a query language only supports operations to retrieve

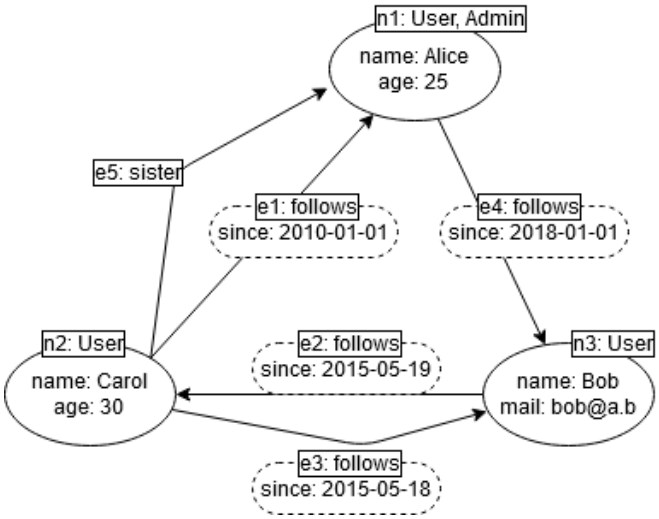


Figure 2.4: A small social graph in the property graph data model.

data. Codd describes such a language more formally as *"a collection of operators or inferencing rules, which can be applied to any valid instances of the data types (of the model), to retrieve or derive data from any parts of those structures in any combinations desired"* [Cod80]. This description limits queries to only retrieve data, and such operators form the class of *Data Query Language (DQL)*. Most definitions extend this strict form to also include manipulation operations that allow to add, modify and remove data. These operations form the class of the *Data Manipulation Language (DML)*. Databases that rely on, or support a schema furthermore include operations of the *Data Definition Language (DDL)*. Analogously to the relational world where the Structured Query Language (SQL) also supports more than just data retrieval operations but is denoted as query language, a graph query language supports DML and sometimes DDL operations as well. Taken together, a graph query language defines operations to access and manipulate data that is structured as a graph and provides support for graph specific operations [ARV19]. [Cha12, Ang12]

Figure 2.5 gives an overview of some graph query languages that are specifically designed for graph data models. As briefly mentioned earlier, the graphical query language G [CMW87] was introduced in 1987 and is one of the first such languages. We will briefly analyze this language as it introduced important concepts that are still in use in modern query languages. G is built on the data model of edge-labeled graphs and introduced the concept of a graphical query. A query $\mathcal{Q}$ consists of a set of labeled and directed graphs, called query graphs. Nodes in a query graph are labeled by either constants or variables, and edges can be labeled by regular expressions over constants and variables. These query graphs can be seen as graph patterns that are matched to subgraphs of the graph database G to evaluate the query.

Figure 2.6 shows an example of a query $\mathcal{Q} = \{\mathcal{Q}_1, \mathcal{Q}_2\}$ that contains two query graphs.

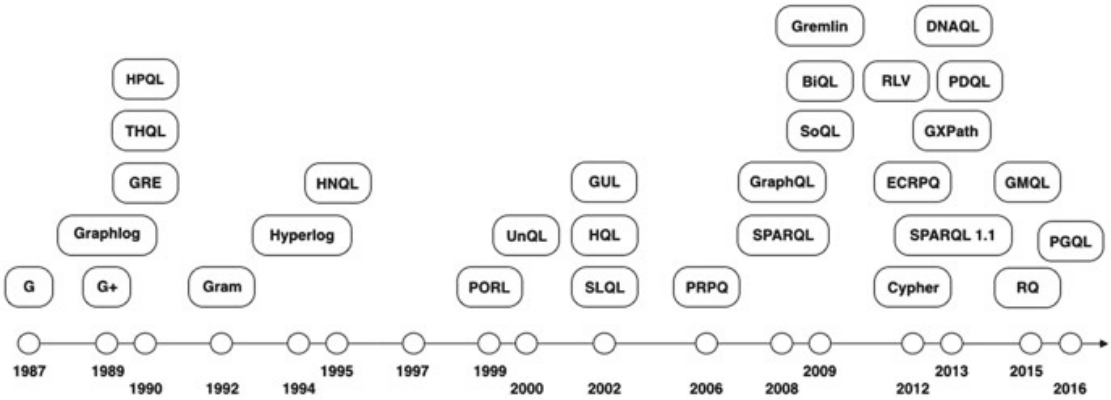


Figure 2.5: Evolution of graph query languages. Source: An Introduction to Graph Data Management, Figure 1.7 [AG18].

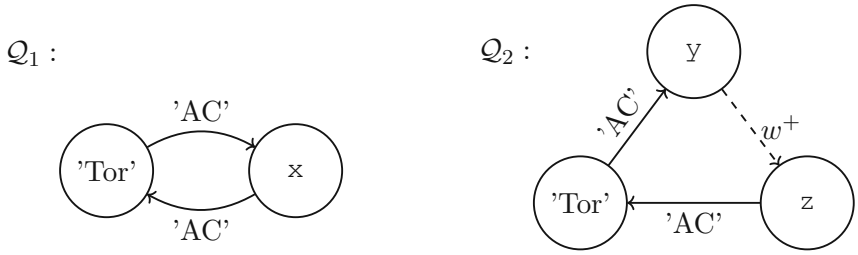


Figure 2.6: An example of a graphical query in the query language G. Source: A Graphical Query Language Supporting Recursion, page 2, example 2 [CMW87].

Each node in the database G represents a city and is labeled by its name while the edges represent airline connections between the cities and are labeled by the name of the airline. The query $\mathcal{Q}$ finds the first and last cities visited by round trips from 'Tor' (Toronto), with the restriction that the first and last connections are via 'AC' (Air Canada). Another restriction is given by the regular expression w^+ that limits all connections between the cities y and z, the first and last cities in a round trip, to be with the same airline. Such a regular expression is depicted as a dashed line to highlight that this is a path and does not have to be a single edge. The support for regular expressions in query graphs allows for a more general and simpler expression of recursive queries compared to other query languages like Structured Query Language (SQL) for example [AG18].

All major property graph query languages introduced in this millennium are built around some core language features. The structure of a query graph as a graphical pattern, like the simple pattern depicted by $\mathcal{Q}_1$, is arguable the most popular feature supported by most, if not all, graph query languages. Depending on the language, these patterns can be either given directly as a graph pattern like in G, or as a textual representation of the pattern as in many modern query languages. We define this pattern and other features of query languages and provide examples for them in Chapter 3. Another notable difference

between many graph query languages and their relational counterpart SQL is, that their input format differs from their output format. Unlike in the relational world, where a query takes a table (relation) as input and outputs a table, a graph query takes a graph as input but in many languages outputs a table containing values [Ang18]. This is the main reason why many graph query languages are not composable. We give more details on the composability of queries in Section 3.6.

The language G evolved into G+ [CMW88] that added support for graph specific functions like the computation of the size of the shortest path, and extended the capabilities of query graphs. Since then, many different graph query languages built on various data models were introduced. Examples are the languages THQL [WS90] and HQL [The02] for hypergraphs, Graphlog [CM90], GRE [Woo90], Gram [AS92] and PDL [ABR13] (as PDQL in Figure 2.5) for edge-labeled graphs and SPARQL version 1.0 [PS08] and version 1.1 [HS13] for RDF graphs. Many of the query languages in Figure 2.5 are more theoretical and not used in real world database system. The rise in popularity of graph databases that mainly use the property graph data model was accompanied by the introduction of query languages built on this model. Examples of such languages are PRPQ [LS06], Gremlin [Rod15], Cypher [FGG$^+$18b] and PGQL [vRHK$^+$16], the latter three being actively used in current graph databases.

Apart from SPARQL, that is standardized by the W3C, there are a variety of languages built on other data models than RDF as no standard exists for these models. Furthermore, at the time of writing, no query language that is currently used by a native graph database has a complete and formal specification. One reason for this is that these query languages have not been around for that long and researchers are only starting to formalize parts of them, like with the popular query language Cypher and its open source variant openCypher [MSV17, FGG$^+$18a]. As many of these query languages are still being actively developed, they evolve and change frequently which makes an effort for a formal specification somewhat impractical at the current time. Another reason for the lack of such a specification could be that querying graph patterns is computationally hard [BLR11]. We will see that querying such patterns is one of the main features of every graph query language in Chapter 3. Therefore, querying graph databases is also computationally hard [Bar13]. This is also one of the reasons for the existence of the multitude of graph query languages that provide different possibilities, functionalities and expressiveness.

The ISO/IEC JTC 1/SC 32[7] is the committee responsible for data management and interchange in the International Organization for Standardization (ISO). Members of their working group 3, that is responsible for database languages, raised the idea of a standalone graph query language that complements SQL in 2017 [MHM17]. This idea was further publicized in the GQL manifesto [Gre18] in 2018 that motivated the move towards the creation of a standardized query language for the property graph data model. National standardization bodies around the world voted in favor of a proposal for such a new language standard, called Graph Query Language (GQL), in 2019 [Gre19b].

[7] https://www.iso.org/committee/45342.html

This was the first time in the last 35 years that the ISO considered the creation and standardization of a new database language [Gre19a].

The project is planned to take four years and even then it will most likely take some time until the vendors of graph databases implement and support the new standard, if they choose to do so in the first place. Therefore, choosing a graph database system before the widespread support of GQL involves not only the properties of the system but also the supported query language. We will now briefly introduce the five graph query languages that we focus on in our comparison.

2.4.1 Cypher

Cypher [FGG+18b] is a high-level, declarative query language that was originally invented for the use in the graph database Neo4j. As that database is one of the most used and well known native graph databases[8], Cypher is regarded as perhaps the most well-known property graph query language [AAB+17]. Its syntax is inspired by SQL to ease the transition for users coming from relational databases and the language is identified as "expressive" compared to other languages [RABM17]. Apart from being inspired by SQL, it also incorporates concepts from other languages like Python and SPARQL as well as from functional programming in general. The development of Cypher came hand in hand with the development of Neo4j and was largely an invention of Andrés Taylor in 2011. Since then, the language has been continually improved and extended. [FGG+18b]

The company behind Neo4j, Neo4j Inc., announced the openCypher project[9] in 2015 that opened up the development of the language and the current version of the language is now referenced as version 9 [ope18]. The project provides an open platform that should enable Cypher to become a fully-specified standard [GJK+18]. This would enable other vendors of graph databases to provide implementations of Cypher in their products. Even though the language is not yet fully-specified or standardized, it has been implemented in other products like SAP HANA Graph, Redis Graph or Memgraph. The move towards standardization came from the goal of the company to establish Cypher "*as the property graph query language*" [FGG+18b]. That company was also behind, or at least involved in, some of the more influential pushes towards the creation of GQL as a new query language standard by ISO/IEC. An influential and driving figure in this area is for example their product manager for Cypher, Alastair Green, who published the aforementioned GQL manifesto [Gre18] among other things. That also explains why some information about the progress of GQL is published on the website of Neo4j, like the notification that the GQL project was approved by the national standardization bodies [Gre19a].

2.4.2 Gremlin

Gremlin [Rod15] is a low-level graph query language that supports both, the declarative pattern matching style but also the imperative graph traversal style [HP16, RWE15].

[8] https://db-engines.com/en/ranking/graph+dbms
[9] https://www.opencypher.org/

This allows the language to be used in any graph computing system, be it an OLTP graph database or an OLAP graph processing system [TPAV19]. Although Gremlin supports both styles, it is more imperative in nature with the main focus on traversing the graph [AAB+17]. Together with the compact syntax used in its queries, it is not that easy to read, especially compared to a higher-level language like Cypher [HP16].

Gremlin was introduced together with the Apache TinkerPop open-source project[10] in 2009 and is still designed and developed by this project [Rod15]. TinkerPop provides a graph computing framework that is not bound to a specific vendor but can be integrated into other database and processing systems. Gremlin is an integral part of this framework and denotes not only the query language, but also the graph traversal machine called Gremlin Traversal Machine (GTM). The GTM processes queries and allows the user to define and use a domain specific query language that is "compiled" to the GTM. Systems that integrate the TinkerPop framework are called TinkerPop enabled. Notable examples of such systems are the Amazon Neptune[11] graph database and the multi-model databases OrientDB[12] and Azure Cosmos DB[13].

As an interesting side node, parts of the TinkerPop framework including the Gremlin APIs originated in the development of the Neo4j database from 2007 onwards [FGG+18b, Lin18]. While early versions of the Neo4j database supported the direct use of Gremlin, this has been removed but is still possible via a plugin like Neo4j-Gremlin[14].

2.4.3 PGQL

PGQL [vRHK+16] is a high-level, declarative graph query language designed and developed by Oracle as part of their Parallel Graph AnalytiX (PGX)[15] framework. The first version of the language was introduced in 2016 and it is still under active development, with version 1.3 [Ora20] being released in March 2020. In contrast to other languages like Gremlin, PGQL is closely aligned to SQL and supports many concepts and keywords from SQL. These are extended by graph specific concepts and algorithms like powerful path expressions and graph pattern matching [AAB+18, AG18]. As in many other graph query languages, the output of a query is structured in tabular form. This allows PGQL queries to be nested inside, and used seamlessly together with SQL queries.

The language is published as an open source specification together with a parsing software[16]. Implementations of the language exist in research projects as a distributed [RTH+17] and non-distributed version [SHvR+16] and it is used in the Oracle (Big Data)

[10] http://tinkerpop.apache.org/
[11] https://aws.amazon.com/neptune/
[12] http://orientdb.com/
[13] https://azure.microsoft.com/services/cosmos-db/
[14] https://github.com/neo4j-contrib/neo4j-tinkerpop-api-impl
[15] https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html
[16] https://github.com/oracle/pgql-lang

Spatial and Graph databases [17]. These databases support the property graph as well as the RDF data model but one has to keep in mind that while in some configurations an in-memory native graph storage via the PGX framework is used, the data is kept in a relational database in other configurations [BPG+19]. Overall, the SQL-like syntax of PGQL tries to lure the SQL community that is already using Oracle products but lacks standardization and support by the broader community and other database vendors [TPAV19].

2.4.4 GSQL

GSQL [Tig] is a high-level query language developed by TigerGraph Inc, the company behind the TigerGraph database. A white paper for the language [WD18] together with one for the database [DXWL19] were published in 2018 and 2019, respectively. Similar to PGQL, GSQL has an SQL-like syntax and goes even further, as a GSQL query that does not contain graph-specific primitives is a standard SQL query. In that sense, GSQL extends the relational SQL to include primitives to query and analyze graphs.

The goal when developing the TigerGraph database was to design "*a property graph database for tomorrow's big data and analytics*" [HLP+19]. To achieve reasonable performance in graph analytics workloads on graphs spanning billions- or trillions of nodes and edges, the database provides computation in massively parallel processing fashion. This objective also influenced the design of GSQL that provides not only primitives for OLTP workloads but also allows for the specification of iterative algorithms and supports the MapReduce interpretation [Wu18]. Another design decision towards better performance, that also came from the tight alignment with SQL, is the use of a schema and type system. In contrast to all other systems and languages we looked at until now, one has to specify the schema of a graph in GSQL's data model. The schema defines the types of vertices and edges in the graph and their possible relationships to each other [Tig]. This allows for a better query optimization, space savings when storing huge graphs as well as more sophisticated security and privacy features like limited access to parts of the graphs for some users [Wu18].

Although the language and the system built around the language is designed to handle and analyze huge graphs, the support for OLTP workloads and unique usage of a schema made us include the language in our comparison.

2.4.5 G-CORE

G-CORE [AAB+18] is a high-level query language designed in a collaboration between industry and academia under the patronage of the Linked Data Benchmark Council (LDBC)[18]. Introduced in 2018, the query language integrates some of the main features from multiple existing languages like Cypher, Gremlin and PGQL in order to provide

[17] `https://www.oracle.com/database/technologies/spatialandgraph.html`, `https://www.oracle.com/database/technologies/bigdata-spatialandgraph.html`

[18] `http://ldbcouncil.org/`

a core for future graph query languages [AG18, ARV19]. As it is a core and reference for future developments, the query language does not support DML or DDL operations. One of the major differences compared to other query languages is that G-CORE is a composable language, meaning that a query takes a graph as input, processes it and outputs a graph as well. This allows queries to be chained or nested, as an output of a query can be directly used as the input of another query. A second major novelty is the treatment of paths as first-class citizens. They can be stored next to nodes and edges in an extended property graph data model and can also have properties similar to nodes and edges. We will analyze and explain this in more detail in Section 3.6.

As G-CORE is a research language, there are only partial implementations in open-source research projects like a parser that is available on GitHub[19].

2.5 LDBC Social Network Benchmark

We analyze these languages based on their characteristics and common language features and compare them by implementing multiple queries in Chapter 4. To pick queries that represent a potential use case in a real world scenario, we use the schema and some queries from the interactive workload of LDBCs Social Network Benchmark (SNB) [EAL+15]. The goal of this benchmark is "*to define a framework where different graph based technologies can be fairly tested and compared*" [AAA+20] and is therefore a perfect fit to test and compare graph query languages. It is developed by an initiative from major actors in industry and academia and currently contains two groups of workloads: the business intelligence workload that focuses on OLAP queries, and the interactive workload with a focus on transactional queries. We limit ourselves mostly to the interactive workload as this one fits our focus on OLTP queries.

The latest stable version at the time of writing is v0.3.2 [AAA+20] and we use mainly this version in our experiments. On one occasion, namely for queries that remove data items, we use the current snapshot of version 0.4.0 [LDB20] as data deletions are not present in earlier versions. The underlying schema as well as all other queries in use did not change between these versions, so we can safely use the new additions together with the model from the stable version.

The interactive workload itself contains queries that are grouped into the following query classes:

- Short Reads (IS, interactive short): This class contains 7 short read-only queries that touch only a small part of the graph.
- Complex Reads (IC, interactive complex): This class contains 14 complex read-only queries that touch a rather significant amount of nodes and edges, most often in the neighborhood of a starting node.

[19] https://github.com/ldbc/ldbc_gcore_parser

- Insertions (IU, interactive update; in the snapshot of version 0.4.0: INS, insert but with the same queries): This class contains insertions of either single edges or nodes together with connections to other, existing nodes. Contrary to the name of these queries in the stable version, interactive *update*, the benchmark does not contain any queries that focus solely on the update of existing entities.

We furthermore use the class that contains deletion queries (DEL), that is now contained in the business intelligence workload. This class is only present in the current snapshot of version 0.4.0 and contains queries that remove either a single edge or a node together with all incident edges and, if needed, nodes that depend on the initially deleted one.

The queries are enumerated inside each class and use the abbreviation of the class as a prefix, i.e. the first query in the class of short reads is IS1. We will state which queries were taken from the benchmark in the next chapter when we introduce queries representing the language features.

2.5.1 Schema

The queries of the SNB are defined on the model of a social network, hence the name. The schema in UML notation can be seen in Figure 2.7. It revolves mainly around persons interacting with one or multiple forums. Tags associated with forums or messages allow groups for specific topics or interests. A person can create or like a message, which in turn can be either a text or an image, but not both. Persons in a forum can further reply to posts or previous comments by replying with a new comment. The schema also supports friendship relationships between multiple persons via the `knows` edge. Although this edge is given as a directed edge in Figure 2.7, the newer version of the benchmark states that it should be treated as undirected, therefore representing mutual relationships between two persons [LDB20]. The schema also allows for further information to be stored, including the city that a person is living in and the university or company they are studying respectively working at.

Apart from the queries, the benchmark comes with a data generator[20] that generates parts of the data statically and parts of it dynamically. This can also be seen in the schema but is of no further concern to us as we focus on the queries working on that model. The generated data using the default parameters represents a snapshot of a potential social network during a period of roughly 3 years with many characteristics taken from real world social networks. As we do not compare the database systems but only the query languages, it is not necessary to generate data and store it in a database. Nonetheless, we did generate and import the data into a Neo4j, a TigerGraph and a Titan[21] database. Titan is a scalable graph database with native support of the TinkerPop stack, therefore supporting Gremlin as a graph query language. This allows us to run the queries from the benchmark as well as implement and test our own queries on these databases and the supported query languages.

[20] `https://github.com/ldbc/ldbc_snb_datagen`
[21] `http://titan.thinkaurelius.com/`

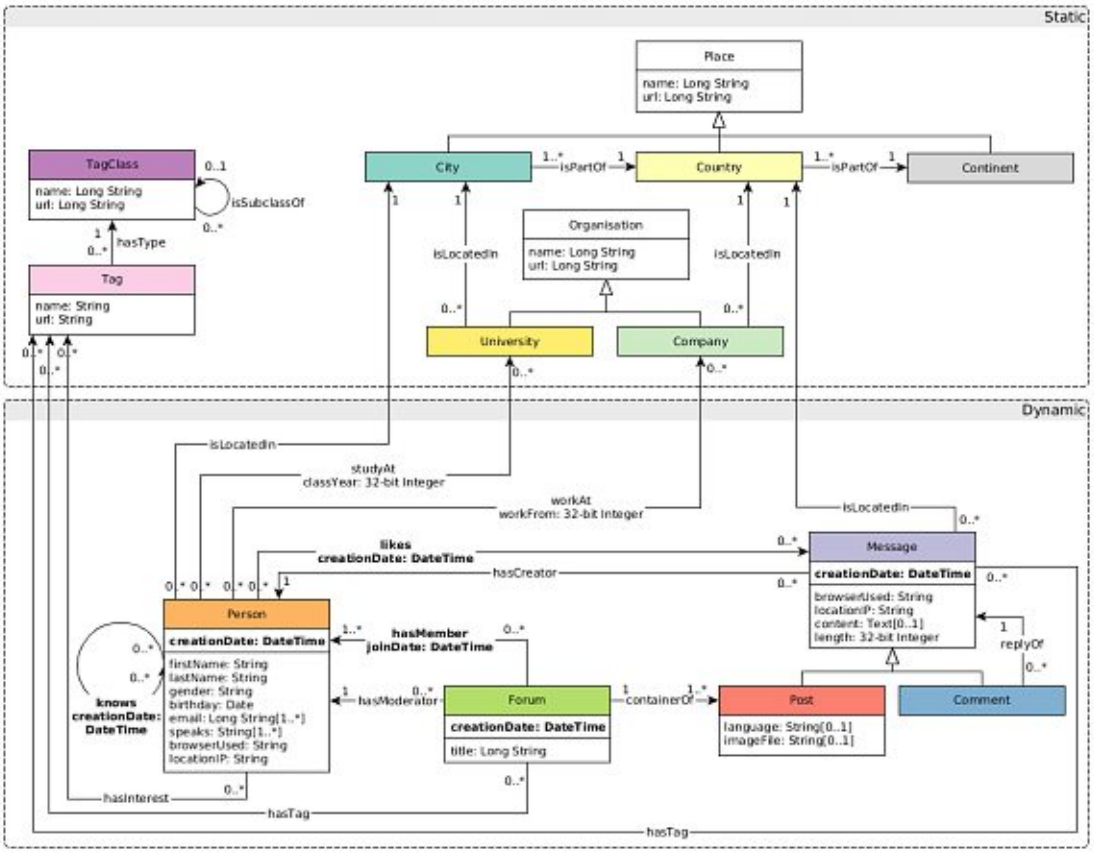


Figure 2.7: The data model in UML notation of the SNB. Source: The LDBC Social Network Benchmark, Figure 2.1 [AAA+20].

CHAPTER 3

Query Language Features

Although there are many graph query languages based on different ideas that vary in their style, expressiveness or purpose, they all share a conceptual core [AAB+17]. This core includes primitives that are based on graph features like graph patterns, paths and neighborhoods (adjacency) [AG18]. As briefly mentioned when we introduced the language G in Section 2.4, *Pattern Matching Queries* form the basis of every graph query language. Apart from them, primitives that allow us to navigate a graph on its paths form a second major feature that we denote as *Path Queries*. We add the feature of *Structure Independent Queries* for queries and primitives that do not consider the graph structure. This is an extension of the summarization queries from Angles [Ang12], that contains simple aggregate functions to summarize the results as well as further operations that do not rely on the graph structure.

As these features form the basis of every graph query language, their underlying primitives are well studied [CM90, BLR11, LV12, Woo12, Bar13, LMV16, TKL19]. We will introduce these groups of queries in more detail and give examples for them. The examples are then implemented in Chapter 4 to compare the languages. We start with an introduction of structure independent and other summarization queries in Section 3.1. Pattern matching and path queries are explained in Section 3.2 and Section 3.3, respectively. We continue with two further features that do not concern the querying part of the languages but their ability to manipulate and potentially define the structure of the data: *Data Manipulation* in Section 3.4 and *Data Definition* in Section 3.5.

3.1 Structure Independent Queries

This feature of a graph query language provides primitives that allow for queries that mainly do not consider the graph structure but focus on the information stored in entities. In its simplest form, a structure independent query operates on the entities (either nodes or edges). It provides primitives that allow the user to select such entities and

their properties as well as aggregate functions that summarize structure independent information. A simple example of such a query in the setting of the LDBC Social Network Benchmark (SNB) would be to select the user with a given first and last name. We do not set the name in the query but expect the user to pass it via the $firstName and $lastName parameters:

Select the person with `firstName`=$firstName and `lastName`=$lastName. (1)

Another example, this time from the benchmark itself, would be the query IS4 (interactive short 4) that selects a message with a given id. The query expects the $messageId as input and returns the content and the creation date of the message. It is more sophisticated compared to the previous one as a message can be either a post that contains an `imageFile`, a post that contains a text-only `content` or a comment with a textual `content`. If there is an image, the result should contain the image file, otherwise the content.

Given a $messageId, retrieve the content of the message and the creation date. (2)

These structure independent queries are not limited to select a single entity, they can for example also return all entities of a given type, like all edges labeled with `knows`, or contain more complicated selection criteria than simple equality.

As briefly mentioned before, the feature also includes summarization queries and therefore primitives that allow us to summarize a graph, or parts of it. These primitives are often applied on the intermediate result of a query that selects parts of a graph, and aggregate the intermediate values to return a single one. Many graph query languages support the same basic aggregate operations as SQL, namely count, sum, max, min and average, and possibly further domain- or graph-specific ones [Ang12]. An example of such a query would be to calculate the average number of spoken languages over all persons:

Retrieve the average number of spoken languages per person. (3)

Most query languages also contain primitives that allow us to compute properties of a graph itself, not the data contained in the graph. Examples for such properties are the number of vertices (order), the diameter, the average degree of a node and so on. In some definitions, like in [Ang12], queries based on such primitives also belong to the group of Summarization queries. However, we do not include these primitives in our definition of this feature as queries using them focus on the structure of the graph and therefore are not structure independent anymore.

3.2 Pattern Matching Queries

Before we start with an explanation of pattern matching queries we will introduce an extension of our small social graph from Figure 2.4 that allows us to go into more detail

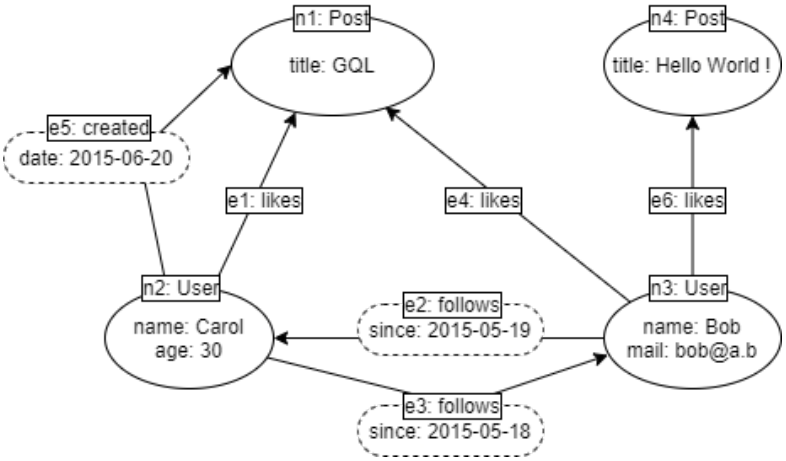


Figure 3.1: A variation of our small social network example including posts in the property graph data model.

in this section. The variation of the network is given in Figure 3.1 where we removed Alice from the network and instead added two posts, i.e. nodes labeled as `Post`, each containing its title. Furthermore, the graph now contains relationships between the users and the posts, given by edges labeled `likes` and `created`.

As mentioned before, matching graph query patterns on a graph form a, if not the, major operation on graphs and therefore are at the core of many graph query languages. Most languages are designed starting with graph patterns and other features are added to these patterns [ARV19]. The idea behind such a graph pattern is to specify the shape of the result via a property graph [1] itself. In difference to the data graph, a query graph is not limited to constants but can also contain variables instead of any constant [AAB+17]. This allows us to query for information and return the values matched by the system. Such a query pattern defines a class of subgraphs [BPG+19]. The system in turn matches the query pattern to the data graph by finding occurrences of these subgraphs that maps possible variables to constants while preserving the structure.

In its simplest form, as explained until now, such a pattern is denoted as *basic graph pattern (bgp)*. As an example of such a bgp we can query for relationships between the post with the title "GQL" and users in our small social network. Figure 3.2 depicts such a query as a possible pattern in a property graph data model.

Many modern query languages do not support queries in a graphical representation but only in a textual one. A Datalog related syntax is often used when studying the theory behind query languages and consists of `(node, edge, node)` triples. Assuming a simpler data model without node and edge IDs, the query could be represented in such a syntax as: `(?W, ?X, GQL), (?Y, ?Z, GQL)` where `W` and `Y` represent node variables

[1] Assuming that the system uses a property graph data model. In other words, such a graph pattern is given in the same shape and style as the data model used by the system and query language.

Figure 3.2: An example of a basic graph query searching for relationships between a specific post and users in a hypothetical graphical query language.

and `X` and `Z` edge variables that are matched against constants in the data graph. Most modern query languages used in industry rely on an ASCII art representation of the query that mimics connections between entities via "−", ">" and "<" signs. The greater and smaller signs are used to specify the direction of a relationship, whereas the lack of these signs describes an undirected relationship in many languages. A possible representation of the query in a Cypher-related syntax looks like this:

```
(w:User) -[x]-> (:Post {title:GQL}) <-[y]- (z:User)
```

In terms of relational operations, basic graph patterns allow for selection based on equality via the use of constants, like `title:GQL`, as well as natural joins via the pattern and the graph structure itself. These basic graph patterns can be extended by further relational operations: union, difference, projection, left outer join (optional) and more sophisticated filtering (selection) rules. The optional operator is especially interesting in most graph databases as it allows us to query incomplete information that is caused by the lack of a schema in many such systems [ARV19]. A query pattern using such extended primitives is denoted as *complex graph pattern (cgp)*. The rising interest in graph databases over the last decade brought the invention of further additions like inexact and approximate matching to graph patterns [CL14, AG18, FPSW19]. However, we limit ourselves to bgps and cgps as mentioned above as these form the core of most modern query languages. Furthermore, additions like inexact matching are still in a research state and not yet supported by any of the major graph query languages. [AAB+17]

We have already seen an example of such patterns in the previous chapter: the query graph $\mathcal{Q}_1$ in Figure 2.6 is a basic graph pattern with a single variable. Furthermore, if we disregard the regular expression in $\mathcal{Q}_2$ for now, the query $\mathcal{Q} = \{\mathcal{Q}_1, \mathcal{Q}_2\}$ forms a cgp using the union operator.

3.2.1 Evaluation Semantics

We will now look at different evaluation semantics in use by the query languages and the systems using these languages as the semantics can influence the outcome of a query. Since cgps are an extension of bgps, their evaluation semantics are also based on the ones from bgps. For this reason, we limit ourselves to bgps in this part. As briefly mentioned before, a bgp is evaluated by matching the pattern onto the data graph such that the variables are replaced by constants while the structure is preserved, at least in some way. This means, that the system searches for all subgraphs in the data graph that are isomorphic to the bgp [AG18] and therefore closely resembles the subgraph isomorphism

problem that is NP-complete [Yan90]. One can identify two major approaches for the evaluation semantics of a bgp: the homomorphism- and the isomorphism-based approach [AAB+17].

- In the *homomorphism-based semantics*, the matches of a pattern on the data graph are not restricted. This means that the evaluation of a bgp $\mathcal{Q}$ on a graph G consists of all homomorphisms from $\mathcal{Q}$ to G and therefore resembles the semantics of queries in relational databases. As it is used in the relational world and heavily studied in theory, this is the most common semantics [ARV19]. However, when using this strategy the database designer has to deal with the potential problem of infinite result sets when enumerating the paths in a pattern as we do not restrict the matches [AAB+18].

- Under the type of *isomorphism-based semantics*, the matches of a pattern are restricted in some sense. This restriction mostly concerns the distinctness of matchings from variables to constants. We can further refine this type of semantics based on the restriction it enforces:

  - If no two variables can be bound to the same data item in a match, we are limited to injective mappings and this is denoted as *no-repeated-anything semantics*. As an effect, the evaluation of a pattern under this semantics preserves the structure of the query pattern as neither edges nor nodes can "collapse" by binding to the same data item.

  - If the restriction only applies to variables on nodes, we call it *no-repeated-node semantics*.

  - Consequently, if the restriction only applies to variables on edges we speak of *no-repeated-edge semantics*.

  Note that the limitations on the no-repeated-node and -edge semantics apply only to variables that map to IDs, i.e. variables for labels or properties can still map to the same data item.

If we evaluate the basic graph pattern from Figure 3.2 on the new variation of our social network given in Figure 3.1, we get the following non-restricted matches:

x1	x2	x3	x4	x5	x6	x7	x8	x9
n2	Carol	e5	created	n1	e4	likes	n3	Bob
n3	Bob	e4	likes	n1	e5	created	n2	Carol
n2	Carol	e1	likes	n1	e4	likes	n3	Bob
n3	Bob	e4	likes	n1	e1	likes	n2	Carol
n2	Carol	e5	created	n1	e1	likes	n2	Carol
n2	Carol	e1	likes	n1	e5	created	n2	Carol
n2	Carol	e1	likes	n1	e1	likes	n2	Carol
n2	Carol	e5	created	n1	e5	created	n2	Carol
n3	Bob	e4	likes	n1	e4	likes	n3	Bob

As the homomorphism-based semantics does not restrict the matches, all of them are valid. Only the first two matches are valid under the no-repeated-anything semantics as the latter ones bind at least two variables to the same data item. Note that we do not only restrict the variables mapping to IDs but all variables, including the ones mapping to labels or properties, in this semantics. An example of this is the third result that is not valid as the variables x4 and x7 are both mapped to the label `likes`. The first four matches are valid under the no-repeated-node semantics since they do not bind variables that map to node IDs to the same ID. As we do not restrict the mapping of edge variables there can be multiple such variables mapping to the same edge label, as is the case in the matches 3 and 4 with the label `likes` on x4 and x7. Therefore, an evaluation under this semantics still preserves the structure of the query pattern as the nodes cannot "collapse", while it does not further restrict the relations between them. Under the no-repeated-edge semantics the first 6 matches are valid as the last three bind the two variables mapping to edge IDs, x3 and x6, to the same ID.

The semantics explained above solely focus on a single match and possibly restrict such a match. This is enough in the setting of bgps where no two such patterns can be evaluated to the exact same match. However, when dealing with the evaluation of cgps and therefore with further operators like union, we can have duplicate matches regardless of the chosen semantics. In that case we can further distinguish between a *set semantics*, meaning that the evaluation of a pattern is accumulated in a set of matches and therefore cannot contain duplicates, and a *bag semantics* that allows duplicates of matches in the result. According to the choice of either set or bag semantics we can for example speak of a homomorphism-based bag semantics or a no-repeated-node-based set semantics. [AAB+17]

Each of the semantics can be desired in some application scenarios and therefore there is no single one used in all database systems and query languages but each system is based on one semantics. While some systems only support a single semantics, others like Neo4j with the Cypher query language allow for query constructs that result in a different evaluation semantics. Still other languages like PGQL even provide keywords that allow the user to switch to a different semantics on a query by query basis.

3.2.2 Queries

Now that we understand the concept behind these queries we will look at some examples from the LDBC SNB. The query IS1 (interactive short 1) represents a simple example of a query using a basic graph pattern.

> Given a $personId, retrieve the first name, last name, birthday, IP address, browser, gender, the creation date of the entry as well as the city of residence of the person. (4)

This query can also be given via the graph pattern in Figure 3.3 that shows the single directed relationship between the two entities in question.

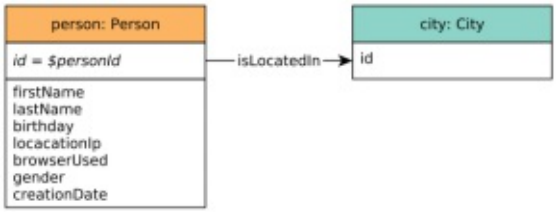


Figure 3.3: Query 4 (IS1) as a graphical query pattern. Source: The LDBC Social Network Benchmark, page 56 [LDB20].[2]

Note that if the query did not ask for the city of residence, it is a structure independent query as the other information is stored directly with the person.

A typical query in the setting of a social network is to search for friends of a person. IS3 represents this in a simple manner where all friends of a given person and the creation date of their relationship are returned.

> Given a $personId, retrieve the id, first name and last name of all their friends as well as the date of the friendship-creation. (5)

Such pattern matching queries can also generate boolean values that indicate whether there exists a possible relationship between some entities. As an example, the query IS7 fetches the author of a given message as well as all persons that created a direct comment to it. The optional relationship, denoted as `knows`, indicates whether these two persons know each other.

> Given a $messageId, get all direct comments replying to this message and a boolean value that indicates if the authors know each other. (6)

[2]The graphical representations of this query and all following ones are taken from the current snapshot of version 0.4.0 of the benchmark [LDB20]. The queries themselves are identical between the stable version 0.3.2 [AAA+20] and the current snapshot, but some graphical patterns in version 0.3.2 do not depict the specified query.

A graphical visualization of this pattern can be seen in Figure 3.4, where the dotted line indicates the flag, or optional relationship.

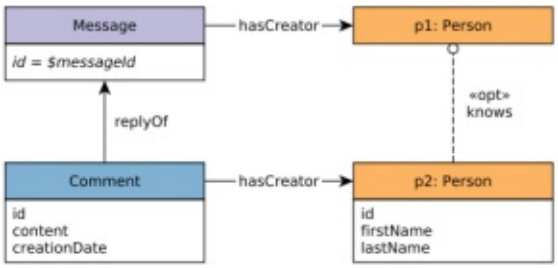


Figure 3.4: Query 6 (IS7) as a graphical query pattern. Source: The LDBC Social Network Benchmark, page 60 [LDB20].

3.3 Path and Navigational Queries

Graph patterns as explained until now allow us to query for bounded and fixed relationships between entities in a graph. However, they are not powerful enough for many queries that navigate more complex relationships in a graph. This in turn is one of the core motivations of using a graph database, as relationships either contain information themselves or tell us something about the entities they connect. Path queries tackle this problem by providing queries that especially focus on one thing: paths. Navigational queries then integrate path queries into patterns to allow for more sophisticated data retrievals.

3.3.1 Path Queries

In its simplest form, a path query can test for the existence of an unrestricted path between two nodes. This is an inherently recursive operation as the database engine traverses from one node to another, often without any bounds on the length of the path. Compared to the world of relational databases, where recursive operations are supported only in a limited way, these operations are heavily used in graph databases and form one of the major advantages when querying related information. Apart from this simplistic form that tests for the existence of an unrestricted path, we can add some criteria that the path has to satisfy. A simple way to achieve this is to allow restrictions on the path by requiring the edges to have a specific label. If we think about a social network, this allows us to search for a path between two users that is limited to friend-relationships (edges labeled with *follows* or *friend* for example). In general, such queries can be grouped as *Reachability Queries* and we will explain further characteristics and ways to specify restrictions on paths in the following part on Regular Path Queries. [AG08, vRHK+16]

Apart from testing for the existence of a path, path queries can also be used to retrieve nodes connected by a path. As an example, this allows us to retrieve all nodes on paths

of the form *friends-of-friends* in a social network, either with a restricted length or even with arbitrary length. Friends-of-friends, or friends-of-a-friend denotes a commonly used query in social networks. Starting from a given node v, the query navigates to all adjacent nodes with the restriction that a traversed edge has to denote a friend-relationship. After the friends of v are explored, it continues to their friends. Note here that, caused by the lack of a definition for this query, it is either used as explained where the result contains all friends and the friends of the friends, stopping at paths of length 2. On the other hand, friends-of-friends can also denote the query that does not stop at paths of length 2 but either at a user-specified length or when the full graph is explored, meaning an unrestricted length. Assuming that we stop after two hops, the query does not only contain the existing friends of a person, but also the friends of their friends, which can be used to suggest new friends or to analyze the network around a person [AAB+17]. More generally, this operation traverses the neighborhood, or adjacency, of a given node. This is an important operation and many graph databases provide dedicated operators for such queries, that are also denoted as *Adjacency Queries* [AG18]. An example of such an operators would be a k-neighborhood one that returns the neighbors up to distance k from a given node [Ang12].

Taken together, Angles et al. summarize the concept behind path queries as follows: *"The idea of a path query is to select pairs of nodes in a graph whose relationship is given by a path (of arbitrary length) satisfying certain properties."* [ARV19]. This can be rewritten to derive a compact form of a path query P given in [AAB+17]: $P = x \xrightarrow{\alpha} y$, where α specifies the restrictions a path between x and y has to satisfy such that the node pair (x, y) is selected. What we lack until now is a notion that allows us to express the conditions on the path, i.e. α.

Regular Path Queries

When looking for possible notations for this, we probably end up with some variant of regular expressions over the set of edge labels, as this form is used in most languages until now [LMV16, AAB+17]. This notation is formalized as a *Regular Path Query (RPQ)* and was introduced in [CMW87] together with the graphical query language G. At its core, an RPQ *"selects nodes connected by a path that belongs to a regular language over the labeling alphabet"* [LV12]. That tells us two things. First, that such a query only ever selects pairs of nodes that are connected by a path. And second, that a node pair is only selected if the concatenation of the labels on a path connecting them form a word in the language of the regular expression [AAB+17]. As a simple example of this, we can write the friends-of-friends query with unrestricted length for Carol in the compact form of a path query as:

$$P = Carol \xrightarrow{follows^+} y.$$

This retrieves all node-pairs `(Carol, y)` that are connected by a path of length at least 1, with the restriction that all edges on the path are labeled as `follows`. Note that we can also use the Kleene Star "$*$" in our query, and, depending on the chosen

evaluation semantics, the query:

$$P = Carol \xrightarrow{follows^*} y$$

can either give the same result as the one from above, or can additionally contain the node pair `(Carol, Carol)`. Furthermore, we can also query for the union of paths via the "|" symbol. As an example, we can search for all posts that are either created or liked by a user that Carol follows as:

$$P = Carol \xrightarrow{follows \cdot (likes|created)} y$$

where the "·" denotes a concatenation.

We can summarize the notation of an RPQ as a query of the form $P = x \xrightarrow{reg} y$, where reg is a regular expression over the alphabet Σ that contains all possible edge labels [LMV16].

The path querying functionality of most modern query languages is based on this notation as it proved to be a good fit to specify restrictions or conditions on the inherently recursive nature of a path [vRHK+16]. As RPQs have been around since the late 1980s and are used in languages until now, the class is well studied, especially also regarding its complexity [Woo12, Bar13, LMV16]. Over time, RPQs have been extended in many ways. An RPQ as explained until now can only traverse edges in a forward direction, but one can also be interested in traversing them backwards. This gives rise to a notable extension, namely the class of the Two-way Regular Path Query (2RPQ) [CGLV00b, CGLV03]. 2RPQs add support for an inverse operator a^- for $a \in \Sigma$ that denotes that the edge labeled a is to be traversed backwards. Let's look at an example of this in our small social network from Figure 3.1. We can query for the friends of all users that liked a post p with the following two-way regular path query:

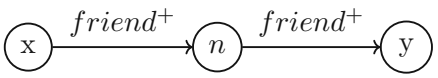
$$P = p \xrightarrow{likes^- \cdot follows} y.$$

Starting from a given node p, the post in our case, the query follows all incoming `likes`-edges in a backward manner and from there all outgoing `follows`-edges in a forward manner.

There are further variants of RPQs that for example add variables [Woo90] or node tests [BPR12] to path queries. We will introduce some of these as we progress with our explanation of navigational queries. As RPQs allow us to express path queries and are used in most modern query languages, at least in some variant, the terms *path query* and *RPQ* are both used to denote this class of queries. The difference is that a *path query* denotes such a query in general, regardless of the underlying variant used by the query language, be it for example a simple *RPQ* or a *2RPQ*. In contrast, the term *RPQ* is more relevant in theoretical literature where it is important to state and define the variant that is used. We will also use this approach from now on and denote the general class of these queries as *path queries* and use the corresponding theoretical terms when we speak about the underlying variants.

3.3.2 Navigational Queries

Early graph query languages, like the language G that we looked at in the previous chapter, allow us to restrict the labels on a path, which is enough to express the friends-of-friends query for example. However, one often wants to express more sophisticated constraints that are not limited to the edge labels on the path alone. With path queries as explained above, we can query for a path between two users in our social network over friends-relationships. If we want to query for such a path with the further restriction that a specific user is visited on the path, path queries are not expressive enough as we have no way to express such an intermediary node. This leads us to *navigational queries*, also called *navigational graph patterns (ngps)*, that allow for a combination of path queries and basic graph patterns. When looking at it from the perspective of a pattern, we are now able to replace an edge in a pattern by a path that is given by a regular expression [AAB+17]. Therefore, we can write the query that requires a specific user n to be on the path as follows:



We have already seen another basic example of such a navigational query when we introduced the language G. The expression w^+ on the query graph $\mathcal{Q}_2$ in Figure 2.6 is a regular expression that is embedded in a basic graph pattern.

As paths in ngps are given as regular expressions over edge labels, and these are nothing else then RPQs, we can also look at ngps from the perspective of regular path queries. From there, ngps are an extension of RPQs and form the class of *Conjunctive Regular Path Query (CRPQ)* [CM90, FLS98]. Queries in that class can deal with intermediate nodes on paths [BLR11], which is exactly what we needed in our motivating example. The term "conjunctive" in CRPQ comes from the fact that by allowing intermediate nodes on a path, we essentially query for the conjunction of multiple paths, each given by an RPQ, that share the intermediary node(s) [LMV16].

Taken together, a navigational graph pattern is a graph pattern where nodes are either constants or variables, and edges can be labeled by constants, variables or RPQs [AAB+17]. Similar to basic graph patterns and the query in Figure 3.2, the patterns in a system built on the property graph model can be further enriched by elements from the property graph. Therefore, we can for example also restrict the nodes to have a specific label, as we did in the example with bgps in said figure.

Further Variants

CRPQs, and navigational graph patterns in general, received lots of attention throughout the last decades as they form the base of many modern graph query languages [TKL19]. This manifested in the theoretical study of this class [CGLV03, BLR11, Bar13] as well as the invention of new variations of it. As explained until now, the addition of path queries is limited to basic graph patterns. Therefore, a natural variation also allows path queries

on complex graph patterns, forming *complex ngps (cngps)*. However, we want to note that this differentiation, and the class of cngps, is not always enforced, most likely since the term ngp often denotes the whole class and all its variations. In the more theoretical terminology of RPQs, we can also add the inverse operator of 2RPQs to CRPQs and derive the class of *conjunctive 2PRQs (C2RPQs*, also *2CRPQs*) [CGLV00a]. U2CRPQs in turn denote a further extension that also includes unions.

However, all of these variations are still not expressive enough for several queries in modern use cases and applications. Under the definitions of RPQs and their variants until now, the evaluation of the query $P = x \xrightarrow{\alpha} y$ on a graph G, $P(G)$, contains the node pairs (x, y) that are connected by a path that satisfies α. However, we cannot output the path or process it, for example by comparing it to others in this evaluation. The class *Extended Conjunctive Regular Path Query (ECRPQ)* [BFL12, BLLW12] tackles this problem by allowing paths to be named, included in the output and compared to others [LMV16]. This essentially raises paths to first-class citizens and as of now seems to be one of the most promising variants for current and future graph query languages. [Bar13]

There is a myriad of further variants using multiple concepts, but, as this is not the focus of the thesis, we will only introduce two more approaches. In all variants introduced until now, we are limited to a single recursion on a path via a regular expression. *Nested regular expressions (NREs)* [PAG10, BPR12], originally developed to query data in the Semantic Web, also allow for another form of recursion. They are an extension of 2RPQs[3] with the addition of a branching operator $\langle\cdot\rangle$ that allows RPQs to be nested. The class *Regular Expressions with Memory (REM)* [LV12, LMV16] on the other hand covers a different use case, namely that of the integration of so called data values on a path. A data value in this case is another word for a property of a node in the property graph data model. An example of this would be the age of a person in a social network. The basic idea is that one can specify when a data value is stored and can than use these stored values. This allows for queries that are not only limited to restrictions of the path via a regular expression, but can also deal with the data values of the nodes on the path. Assume that we are interested in a variation of the friends-of-friends query where we only traverse to friends of the same age. With REMs, we are able to store the data value (age) of the first node on the path, then traverse an outgoing friends-relationship, read the stored data value and compare it to the new one.

3.3.3 Query Evaluation: Semantics, Output and Complexity

Now that we have introduced path and navigational queries we will take a closer look at their evaluation in general as well as some semantics, possible outputs of such queries and finish with a note on the evaluation complexity. We limit ourselves to path queries in this section as the evaluation of navigational ones can be derived from this, together with the evaluation of bgps (and possibly cgps).

[3] Note that they are a class of path queries in their original form, therefore there are no patterns in these queries.

Query evaluation and semantics

As mentioned above, the evaluation of an RPQ $P = x \xrightarrow{\alpha} y$ on a graph G contains the node pairs (x, y) that are connected by a path that satisfies α. ECRPQs on the other hand also allow us to retrieve paths as a result of a query. From the practical perspective of a database system however, the system has to find all paths that match a given pattern before it can project to the desired elements. As an example, let's look at the evaluation of the RPQ P from above. In a first step, the system has to find all paths in the graph that satisfy α, and can then project to the start and end nodes of these paths to get (x, y) as output. Therefore, when looking at the evaluation semantics of path queries from a practical perspective, it does not make a difference whether the query language allows the output of paths or not. To explain the evaluation semantics in this section, we will look at the paths that are found during the evaluation of a query in a database system.

Before we can explain the evaluation of path queries, we need a formal definition of a path and its label in a graph G. A path π is a sequence $n_0e_1n_1e_2 \dots n_{k-1}e_kn_k \mid k \geq 0$ where each e_i is an edge between the nodes n_{i-1} and n_i. As briefly mentioned before, the evaluation of a path query contains only paths whose concatenation of the edge-labels form a word in the language of the regular expression. $Lab(\pi)$ denotes the label of π, given as $Lab(\pi) = a_1a_2 \dots a_{k-1} \in \Sigma^*$, where each a_i denotes the label of e_i and $a_i \in \Sigma$ [4]. The evaluation of a path query $P = x \xrightarrow{\alpha} y$ on a graph G, denoted as $P(G)$, contains all paths in the database whose label $Lab(\pi)$ satisfies α. Note that our definition of a path also allows for paths of length 0 that contain only the node n_0. If we use the regular expression that accepts all words over Σ, Σ^*, in an RPQ $P = x \xrightarrow{\Sigma^*} y$, we do not impose any constraint on the path. Therefore, the evaluation of this query also contains the empty path with $k = 0$ and the single node n_0. [AAB+17, ARV19]

This definition allows us to evaluate path queries on graphs but also leads to a potential problem: the evaluation can result in an infinite number of paths. As an example, we can look at an extension of the friends-of-friends query (with unlimited length) where we are not interested in the transitive closure of the friends but in the posts that they liked. Starting form a fixed node $n2$ (the user Carol), this query can be written as:

$$P = n2 \xrightarrow{follows^+ \cdot likes} y.$$

If we evaluate this query on the graph G from Figure 3.1, we get an unlimited number of paths caused by the cycle between Carol and Bob. The evaluation $P(G)$ contains, among others, the following paths that are given by their nodes and edges as well as their label:

[4] As a reminder, Σ contains all possible edge labels.

π											$Lab(\pi)$
n2	e3	n3	e4	n1							follows·likes = follows^1·likes
n2	e3	n3	e2	n2	e1	n1					follows^2·likes
n2	e3	n3	e2	n2	e3	n3	e4	n1			follows^3·likes
n2	e3	n3	e2	n2	e3	n3	e2	n2	e1	n1	follows^4·likes
			⋮				⋮				⋮

Similar to the different semantics when evaluating graph patterns, there are multiple ones for the evaluation of path queries that differ mainly in their handling of duplicates in the evaluation $P(G)$. We will now describe four common evaluation semantics in use by graph query languages [AAB$^+$17].

- Under *arbitrary path semantics*, the paths in $P(G)$ are not restricted in any way. Therefore, all paths that satisfy the restrictions of the path query are present in the evaluation. If we look at our example query above, the evaluation contains an infinite number of paths caused by the loop in the original graph that satisfies the condition in P. As it is not feasible to enumerate an infinite number of paths, one solution is to restrict them in some way. In some scenarios we are only interested in whether there is such a path or not, and the system can stop as soon as the first such path is found. Another possibility is to return not the paths, but the pairs of nodes that are connected by them, which results in a finite number of node pairs. This results in an output as defined for RPQs. There are further possibilities to deal with this problem but the main takeaway is that $P(G)$ may contain an infinite number of paths under such an unrestricted semantics.

- The *shortest path semantics* restricts the paths in $P(G)$ to be shortest paths only. That means, only paths of minimal length between any two nodes that satisfy the conditions in P are contained in $P(G)$. In the evaluation above, only the first path is valid under this semantics as all further ones are not paths of minimal length between the nodes $n2$ and $n1$.

- In case of the *no-repeated-node semantics*, a path that satisfies the conditions in P is only included in $P(G)$ if no node appears more than once on the path. These paths are also known as *simple paths* and the evaluation of RPQs under this semantics has been studied extensively [MW89, ACP12, LM13], which cannot be said for many of the other semantics. Using this semantics, only the first path given in the table above is valid as at least one node appears more than once on the others.

- The evaluation under the *no-repeated-edge semantics* contains only matching paths where no edge occurs more than once. In our evaluation of $P(G)$ above, only the first two paths satisfy this as the latter ones repeatedly take the same route.

Output

Now that we have seen some evaluation semantics for path queries, we will briefly look at different forms of possible outputs for such queries. Again, we look at this from a practical perspective, where a user may for example only be interested in a simple true/false-answer and not in nodes, properties or paths. In such a case, it is enough to output a boolean value. As an example, the query $P = a \xrightarrow{\alpha} b$ with given nodes a and b asks whether the nodes are connected by a path satisfying α and can be answered by a boolean value depicting whether $P(G)$ is empty or not. In other cases, the start and end nodes (a and b in the previous example) are not given but we are searching for them, i.e. we want to find the nodes connected by a path satisfying α. The output of such a query contains these node pairs and therefore coincides with the definition of the output of RPQs.

Apart from these outputs that have a fixed arity, we could also be interested in the paths themselves which are inherently of variable length. As we have seen before, including all paths that satisfy a query under the arbitrary path semantics can result in a large, and potentially infinite, number of paths. Furthermore, a database system offering such outputs also has to deal with paths of potentially unlimited length. One approach to get a compact representation of the output is to return a graph consisting of the paths that are contained in the evaluation of the query. This in turn would allow for query composition as a query takes a graph as input and would output a graph as well. We will look at this in more detail in Section 3.6. However, this is not supported by any modern database system or graph query language except in the research language G-CORE as of now. Generally, the representation of paths in the output of a query differs from system to system as there is no consensus for this until now. [AAB+17].

Complexity

Evaluating a path query on a graph is a complex undertaking in general. Caused by their recursive nature, the system often needs to load, or at least go through sizable data space. For example, a search for all node pairs connected by a path that satisfies a regular expression under the simple path semantics (no-repeated-node) is NP-complete in the graph size [MW89, Bar13]. This has led to the development of many semantics based on the arbitrary-path one, that for instance stop as soon as one path is found. Barcelò showed that such a semantics leads to tractable complexity when evaluating RPQs [Bar13]. However, this does not mean that all modern query languages and database systems use a variant of arbitrary path semantics as we will see in our analysis. [AG18, AAB+18]

3.3.4 Queries:

Now that we understand the concepts and evaluation semantics of navigational queries, we will look at some examples from the LDBC SNB. In our explanation of path queries, we started with reachability queries that test for the existence of a path between two

nodes. We also mentioned the possibility to constrain such a path, for example by only allowing edges with a specific label. The query IC13 (interactive complex 13) is an example of that, with the addition that we do not only test for the existence of such a path, but are also interested in the path itself and the length of it.

> Given two persons by their id, find the shortest path between them in the subgraph induced by the `knows` relationships. Furthermore return the length of the path, 0 if both ids are equal and -1 if there is no such path. (7)

Figure 3.5 depicts the query as a navigational graph pattern that uses a slightly different notation for the regular expression $knows^*$, namely "$knows^*0..$". The "*" signalizes a path of some length where 0.. gives the lower bound. Likewise, $knows^*1..3$ would denote a path of length at least one and maximum three, where all edges have to be labeled with `knows`. Note that we are not able to specify that we are only interested in the

Figure 3.5: Query 7 (IC13) as a graphical query pattern. Source: The LDBC Social Network Benchmark, page 54 [LDB20].

shortest path in this representation. The same limitation applies to the representation as an RPQ:

$$P = person1 \xrightarrow{knows^*} person2.$$

In our introduction of path queries, we then moved on to adjacency queries that explore the surroundings of a given node. Query IC1 is an example of such a query that examines the adjacency of a person.

> Given a person, find persons with a given first name in their adjacency over `knows` relationships of at most 3 steps. Return these persons, their distance as well as summaries of their workplaces and place of studies. (8)

Figure 3.6 depicts this query as a navigational pattern. As an advantage of such a representation, we can get a feeling of the difficulty of a query in terms of accesses to different elements, like the ones on `City`, `Company` and `Country` here.

Now that we have seen an example of both, a reachability and an adjacency query, we will look at general navigational query. Query IS2 gives us the last messages created by a user, the original posts in the conversations as well as the persons that created these posts. An application could then use this information to analyze relationships between these persons for example.

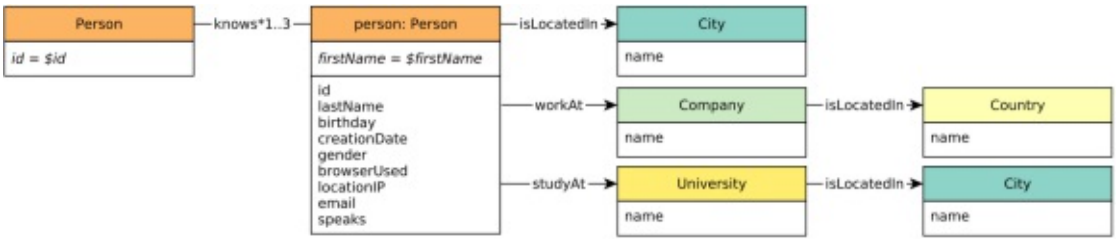


Figure 3.6: Query 8 (IC1) as a graphical query pattern. Source: The LDBC Social Network Benchmark, page 42 [LDB20].

Retrieve the last 10 messages created by a given person. For each of those messages, return that message, the original post in its conversation as well as the author of that post. (9)

This is a navigational query as we have a path of variable length between the message and the original post in its conversation. What makes this query a bit challenging is the fact that a message can also be a post. In this case, the original post of the conversation will be the same message and should occur twice in the output. Figure 3.7 depicts this query as a navigational pattern.

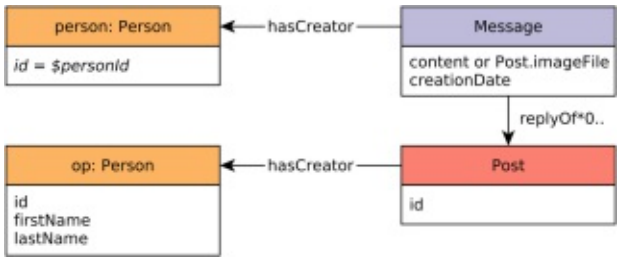


Figure 3.7: Query 9 (IS2) as a graphical query pattern. Source: The LDBC Social Network Benchmark, page 57 [LDB20].

Summary:
These features, namely graph patterns, paths and their combination in navigational queries form the core of most, if not all, modern graph query languages. In all example queries that we looked at until now, we used queries that retrieve data from a given graph. Such queries form a sub-language of a general query language that is denoted as DQL. As briefly mentioned in the introduction of query languages in Section 2.4, a general query language also contains operations to manipulate the data and sometimes even the schema. These sub-languages are in turn denoted as DML and DDL and we will now look at them, their underlying mechanics and potential problems.

3.4 Data Manipulation

Queries that manipulate the data allow us to change the graph by adding nodes or edges, but also by removing or updating existing ones. These graph transformations are specified in a query using primitives that are based on the previously introduced patterns and paths [AG18]. Therefore, queries that manipulate the data are expressed similarly to queries that retrieve data as both rely on graph-oriented operations [AG08].

In order to update or remove an entity, we first have to identify it. This can be problematic in some scenarios. Think about a social network that can contain multiple people with the same first and last name. To avoid an identification problem, many graph databases use globally unique IDs that are associated to each entity, similarly to the world of relational databases. Therefore, if one wants to remove a person with the name of "John Doe", we could first query for all such persons, select the ID of the one that we want to delete and then send the DML query to remove the entity.

Another thing that we have to keep in mind when removing nodes are their relationships to other nodes. When we remove such an entity, all incoming and outgoing edges become invalid as they are left with only one adjacent node. It is up to the system to either remove these stale relationships together with the node or to manipulate them in order to achieve a valid graph again. There is no correct behavior for all scenarios. While some systems remove such edges with the node, others provide explicit operators to either delete only the node or also the adjacent relationships and there are even proposals for algorithms that can deal with stale edges [SW14]. In some scenarios, the removal of a node could also imply that other, often adjacent, nodes should be removed. Let's look at the relationships between a forum and the posts in this forum in the data model of the SNB in Figure 2.7. If we are to remove a post, all comments that reply to this post hang in the air. Similarly, removing a forum renders all posts in that forum to be without a container.

Apart from the removal of entities we can also insert and update entities using such queries. This allows for a dynamic change of the data itself and, assuming that the database does not use a schema as is the case for some graph databases, also the data model. Let's look at our small social network where a node is identified as a person only by its `User` label. If we update such an entity and change the label to `Person`, we are left with a data model containing both, nodes labeled as `User` and `Person`. This gives us a glance at the extreme flexibility of such systems as one is not bound to any schema. On the other hand, querying a completely schema-less graph becomes nearly impossible as for example persons can be labeled as `User, Person, Man, Woman ...`. This problem does not only concern node labels but also edge labels and properties in property graphs as we could store the name of a person in a single property or split it up into multiple ones for example. That means, that even if the database system does not enforce a schema it is in the interest of the users to agree on a set of labels and properties and to use them throughout the database. The flexibility that remains then is the adaptability to future changes where a new label can be seamlessly added to either

existing entities or new ones and can then be used in future queries. We do not have these problems in systems relying on a schema as the schema defines the available labels and potentially also the properties. However, we have to adapt the schema before we can introduce a new label, property or entity.

3.4.1 Queries:

Now that we have highlighted some of the potential problems and types of queries that are possible with this sub-language, we will look at a few examples.

In the setting of our social network, adding a new person to the graph is an important operation. However, we do not only want to add a node containing the information about the person but also relate it to other entities in a single query. The query INS1[5]does exactly that by adding a node as well as edges depicting their location, interests and so on.

Add a node denoting a person and connect it to existing nodes depicting their location, interests and place(s) of work and/or study. (10)

Figure 3.8 depicts this query as a graphical pattern and the various parameters (preceded by a dollar-sign) that will either be set as properties or are needed to build the relationships.

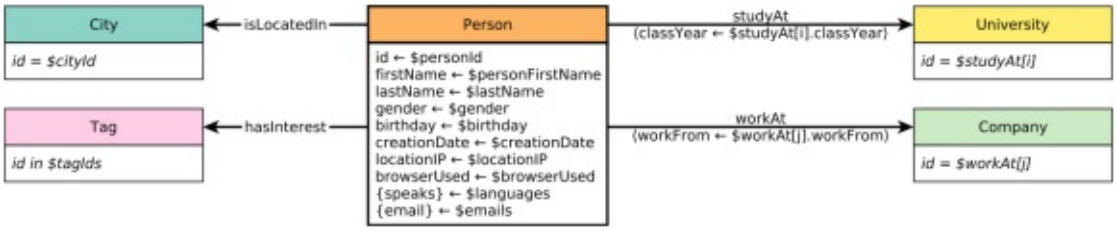


Figure 3.8: Query 10 (INS1, IU1 in [AAA+20]) as a graphical query pattern. Source: The LDBC Social Network Benchmark, page 61 [LDB20].

As an example of a query that deletes one or multiple entities we have to look at a snapshot of version 0.4.0 as deletions are not present in earlier version. The query DEL7 removes a comment together with the corresponding relationships as well as all further comments that reply to the initial one.

Delete a comment, its incident edges and recursively all comments replying to the intitial one. (11)

[5]Remember: the insertion queries INS1-INS8 in the snapshot of the most recent version 0.4.0 [LDB20] equal the queries IU1-IU8 (interactive update) in the stable version 0.3.2 [AAA+20].

Figure 3.9 depicts this query as a graphical pattern. We can see that the removal triggers the invocation of a recursive delete on the replying comment(s). This in turn deletes also their edges until there are no dangling comments left. Therefore, this query potentially touches many entities and, depending on the level of support for such transitive removals from the system and the query language, can be quite sophisticated.

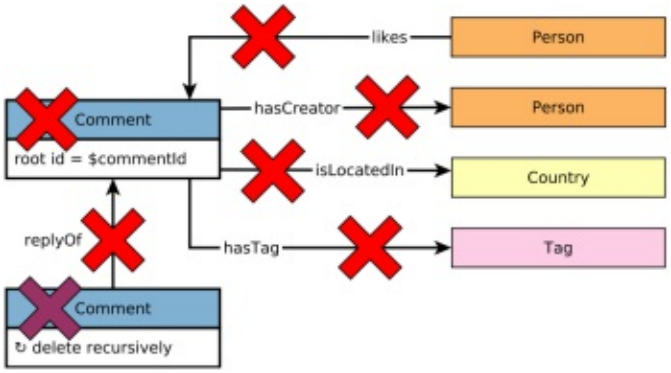


Figure 3.9: Query 11 (DEL7) as a graphical query pattern. Source: The LDBC Social Network Benchmark, page 87 [LDB20].

The SNB does not contain queries that update an already existing entity, be it a node or an edge. To this end, we include Query 12 that updates the property of an edge, namely the year a person graduated at a university.

Update the year (`classYear`) a given person graduated at a given university. (12)

3.5 Data Definition

When we introduced the different data models in Section 2.3, we highlighted that some of them support a schema while others do not. An example of such a schema can be seen in the RDF data model in Figure 2.2. We also mentioned that the widely used property graph data model does not support a schema out of the box. However, even if a database system and the underlying data model do not support a schema, it is in the interest of the users to agree on some sort of a schema as we have motivated in the previous section. Therefore, many graph databases allow, or even require, the specification of some sort of schema or restrictions on the graph, even if they are built on the property graph data model.

Using a strong type system where labels, properties and their data types as well as possible relationships are specified in the schema limits the flexibility. This loss is traded for performance as a schema can be used to optimize query execution, improve access to entities or even reduce the space needed to store them [DXWL19]. Apart from performance improvements, a schema also allows for security and privacy features

[WD18]. One can limit for example the access to parts of the graph for certain users and hide other parts from them. In applications dealing with data in a known and structured form, using a pre-defined schema representing this structure does not limit the flexibility while providing the aforementioned performance and security benefits. Therefore, we cannot derive a single best setting regarding the use of a schema for all applications.

The varying trade-offs between the flexibility and having at least some sort of schema are one of the reasons for the existence of the various graph databases and often a major differentiation between them. Even if we limit ourselves to databases and languages relying on a schema we end up with multiple variants of such schemata. Where one language for example allows an edge to be related to its reverse edge as in GSQL, we cannot express this relationship in other languages like Cypher.

While DML queries allow us to manipulate the data graph, Data Definition Language (DDL) queries are used to manipulate the schema. This includes the initial creation of the schema as well as adaptions later on.

3.5.1 Queries:

The SNB does not contain any queries that manipulate the schema as it assumes the pre-defined model from Figure 2.7. In our comparison of the query languages we will describe whether a language supports or relies on a schema as well as the type of the schema. As an example of such a DDL query we try to add a street to our data model in Query 13, meaning that we adapt it such that a person is not directly located in a city but in a street which in turn is located in a city.

Add a street entity to the data model and change the relationship
`Person[0..*]-isLocatedIn->[1]City` to
`Person[0..*]-isLocatedIn->[1]Street[0..*]-isLocatedIn->[1]City`. (13)

3.6 Further Features

The first three features that we introduced, namely structure independent, pattern matching and navigational queries, are the major building blocks used in many graph query languages. These features allow us to query data and are also used in DML queries to modify it. In a database that supports a schema, DDL queries are used to modify it. Depending on the representation of the schema, they can also be built on top of these three features as one could for example specify a schema using a pattern. Apart from these features that form the basis and group possible queries, more general features of graph query languages have been identified and we will now look at two of them. These two features are more a characteristic of a language and tell us something about its

expressiveness in certain use cases. The LDBC Technical User Community identified these as two of the major issues in existing graph query languages [AAB+18].

- **Composability**

  The first one, composability, denotes that the output or result of a query can be used as input to another query in the same language. This allows for nested subqueries as in SQL but also for a linear composition of queries where the first part of a query selects some data which is then passed on as the input to a second query. A composable query language encourages a modular approach and interoperability between queries as we can plug one query into another one. This in turn allows for an abstraction of problems as we can decompose bigger queries into smaller pieces where the inputs and outputs are compatible [AAB+18].

  The input is only compatible to the output if they have the same, or at least a compatible, data model [ARV19]. As briefly mentioned in the introduction of query languages, a graph query takes a graph as input but in many languages outputs not a graph but a table containing values [Ang18]. Therefore, queries are not composable in these languages. The paper that introduced the research language G-CORE goes even one stop further and states that *"current query languages do not provide full composability because they output tables of values, nodes or edges."* [AAB+18]. Following this argument, a graph query language is fully composable if it outputs the result of a query as a graph. Unfortunately, as of now this is not supported by any of the current graph query languages except by G-CORE itself.

- **Paths as first-class citizens**

  The second feature deals with the representation of paths in the data model. We already mentioned this feature when we introduced the class of ECRPQ in Section 3.3.2. Paths are raised to first-class citizens for ECRPQs as they can be named, included in the output and compared to others [LMV16]. An entity in any programming language is said to be a first-class citizen if it supports the operations that are generally available also to other entities. If it cannot be returned or stored however, it is reduced to a second-class citizen [Sco06]. Consequently, raising paths to first class citizens allows us to add labels and properties not only to nodes and edges but also to paths [AAB+18]. These changes require a more sophisticated data model, as the models explained until now cannot deal with stored paths.

  As of now, the feature is again primarily supported by the research language G-CORE. To achieve this, G-CORE uses an extension of the property graph data model, namely the *Path Property Graph (PPG)* data model, that allows paths to be stored and enriched with properties and labels. Figure 3.10 depicts a small social network in this model where node labels are depicted as shapes and edge labels as lines with either an arrow or rhombus. The novelty in this data model lies in the lower left part where we find two stored paths, one from John to Alice and one from John to Celine. As the users on the start and end node on both of these

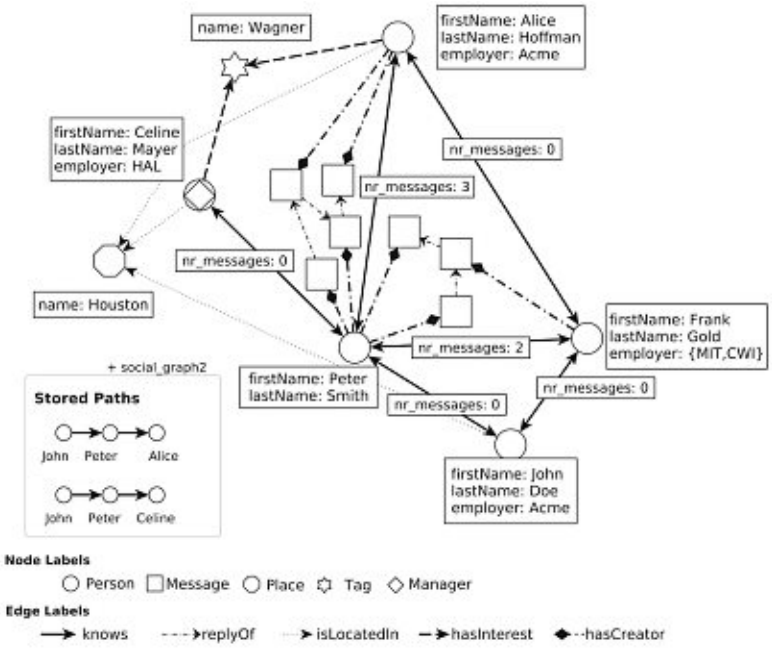


Figure 3.10: A small social network in the path property graph data model that includes stored paths. Source: G-CORE A Core for Future Graph Query Languages, Figure 5 [AAB+18].

paths live in the same city, namely Houston, we could label them for example by `sameCityRelation`. Furthermore, we can store the length of such a path as a property of it such that a query can directly access this attribute without needing to compute it.

CHAPTER 4

Query Language Analysis

Now that we have introduced the features that form the conceptual core of all graph query languages, we analyze 5 contemporary languages based on those features. The 5 languages are: Cypher as it is perhaps the most well-known property graph language, Gremlin for its imperative approach, PGQL for its powerful path patterns, GSQL for its use of a schema and natively parallel approach and G-CORE, that builds upon the other languages and offers full composability. We did not only choose those languages for their different features and approaches, as nearly every major contemporary graph database supports at least one of those query languages.

To achieve a comprehensible comparison, we analyze the languages one after another. For each language, we start with a general introduction that includes the core principles of the language and give examples for the structure of a query. We then analyze the languages in depth based on the previously identified features. We go through the implementations of the queries from Chapter 3, highlight differences to other languages and look at the expressiveness of a language by the support for specific functions as well as potential shortcomings. Although we implemented all queries from the previous chapter in the 5 languages, we limit ourselves to those that are different to other implementations in this thesis. Furthermore, the query definitions in the SNB expect an implementation to rename each variable in the result. We omit this for brevity on most queries in this chapter as this does not further our cause of comparing the languages. Nonetheless, a full implementation of all queries can be found on our GitHub repository[1].

[1] https://github.com/martin-kl/Diploma_Thesis_01526110_code

4.1 Cypher

Cypher is perhaps the best-known query language for property graphs and focuses heavily on patterns. The language is used in multiple databases and projects, with the version in the Neo4j database [Neo20] and the one from the openCypher project [ope18] being the most influential ones. As the language is not yet standardized, it is still subject to change and the versions and implementations of it differ in some areas. Unless specifically mentioned, the concepts and functions apply to all current implementations of the language, especially also Cypher 9, the current version of openCypher, and the version in the Neo4j database. [AAB+17]

Whereas programming languages like Java and other graph query languages like Gremlin are imperative in nature, meaning that they focus on the description of the program's operation and in terms of query languages on how to compute the result, Cypher is a declarative language. Therefore, it focuses on what a query should retrieve instead of how it achieves the result. This is one of the areas where Cypher is similar to SQL, that is declarative as well. Apart from that, many keywords in Cypher are inspired by SQL to ease the transition for users coming from the world of relational databases. Being a declarative query language and using many of the same keywords as SQL, it comes only natural that the general structure of a Cypher query resembles the one of an SQL query. A query in both languages consists of clauses that are chained together [ope18]. In contrast to SQL however, Cypher queries are conceptually structured linearly. Therefore, we can think of the execution of a query as starting with the evaluation of the first clause by generating a result for it, which in turn is passed on as input to the second clause and so on. This linear combination leads to another difference as it does not allow the projection (the `RETURN` statement, Cypher's variant of SQL's `SELECT`) to be at the beginning of the query as first clause, it has to be at the end of it. [FGG+18b]

Algorithm 4.1 depicts a simple Cypher query consisting of three clauses, one in each line. The `MATCH` clause is the entry point of a query and contains the basic graph pattern (bgp) that the query is searching for. Together with the `WHERE` clause that further restricts the pattern, this query searches for relationships between the person John Doe and other nodes. Parentheses "`( )`" in the pattern depict a node whereas edges are depicted by "`-[ ]-`". As we can see in our example, these brackets can contain either a single word that denotes a variable (`rel, dst`) or a variable together with a filter on a label that is given after the colon. `(pe:Person)` for example matches only these nodes to the variable `pe` that are labeled as `Person`. [AAB+17]

```
1 MATCH (pe:Person) -[rel]- (dst)
2 WHERE pe.firstName='John' AND pe.lastName='Doe'
3 RETURN p, rel, dst
```

Listing 4.1: A Cypher query that selects the neighborhood of the person John Doe.

We can also see the conceptually linear structure of a query and observe that it can be evaluated one clause after the other, starting from the topmost that selects all such

patterns, followed by the restriction on the variable p to the person John Doe, to the projection in the RETURN clause. However, a system implementing the query language can re-order clauses as long as the semantics of the query is retained [FGG+18b]. Therefore, the linear structure holds only conceptually and an implementation is not bound to the order of clauses when evaluating a query. That allows an implementation to use the restrictions given in the WHERE clause already when searching for patterns in the MATCH clause, therefore improving the evaluation efficiency. The strong dependence between the WHERE and MATCH clauses lead to a short form that allows a restriction on properties also in the pattern. The query in Algorithm 4.2 is equivalent to the previous one as it contains the restrictions on properties in the "{ }" brackets. [ope18]

```
1 MATCH (p:Person {firstName: 'John', lastName: 'Doe'}) -[rel]- (dst)
2 RETURN p, rel, dst
```

Listing 4.2: Query 4.1 in a more compact notation.

Now that we have seen a simple example of a query, we will look at the evaluation of queries and clauses and the information passing between clauses in more detail. A clause resembles a function that, analogously to SQL, operates on an input table and produces a table as output. In other words, each clause gets an input table, applies a function on the values in that table, and produces an output table that in turn is the input table for the next clause. These tables are also called *binding tables* in Cypher as they bind variables from the query to elements in the graph or other values. However, the definition of a query consisting of clauses that operate on these tables leads to a problem, as a Cypher query in an abstracted sense outputs a table but takes a property graph as input. We can generate the output as a table under the definition of a clause, but until now each clause expects a table as input as well. That is especially also true for the first clause in a query, meaning that such a clause gets an empty table as input. The MATCH clause is the main representative for a special group of clauses regarding these binding tables [2], as they usually take an empty table as input and can populate it with entries from the property graph. Therefore, we need such a clause in every query that interacts with the graph, even if we simply select all nodes in the database as does the query in Algorithm 4.3. This also highlights the status of patterns as an integral part of Cypher. [FGG+18b]

```
1 MATCH (n)
2 RETURN n
```

Listing 4.3: A simple Cypher query that selects all nodes.

Linear Composition and the WITH clause

Apart from the conceptually linear flow between clauses in a single query as seen until now, Cypher also supports the WITH clause that allows for a linear composition of queries. As an example, the query in Algorithm 4.4 starts at John Doe and in a first step traverses

[2] If we restrict ourselves to the DQL, MATCH is the only such clause, but there are others like MERGE that are part of the DML.

to all persons m connected to him. Since we are only interested in the person with the alphabetically last first name, we have to order them by their first name in descending order before we can select the top result. We then want to pass that person as input to the second part of the query, namely the second MATCH. However, ORDER BY is a sub-clause in Cypher that has to follow a projecting clause, meaning either RETURN or WITH. To this end, we have to use the WITH clause to specify which variables are passed on from the binding table of the first part of the query to the second MATCH. Furthermore, only these variables can be used in the ORDER BY clause, that in turn is evaluated before the second MATCH as it is a sub-clause of WITH. We highlight this dependence of ORDER BY on WITH in the algorithm by the indentation of line 3, which is however not required or motivated by the Cypher guidelines and should only help the reader in this case. The second MATCH clause in the query now uses both, the input variable m it got from the previous part and the property graph, to populate its binding table with entries for the variable o and return properties of them. [ope18]

```
1 MATCH (p:Person {firstName: 'John', lastName: 'Doe'}) -[rel]- (m:Person)
2 WITH m
3   ORDER BY m.firstName DESC LIMIT 1
4 MATCH (m) -[r]- (o:Person)
5 RETURN o.firstName, o.lastName
```

Listing 4.4: A Cypher query that contains a linear composition of queries via the MATCH clause.

Grouping

One well-known keyword from SQL however is missing in Cypher, namely GROUP BY. The functionality is instead provided directly by the RETURN clause, as each non-aggregate function is implicitly taken as key to group by. This can be seen in Algorithm 4.5 where n becomes the grouping key and we count the number of neighbors o.

```
1 MATCH (n) -[rel]- (o)
2 RETURN n, count(*) LIMIT 10
```

Listing 4.5: A simple Cypher query showing an alternative to SQL's GROUP BY.

Apart from grouping in the last clause, we can also use WITH to specify the group(s) and the aggregate operation. This allows us to work with the result of the aggregation, for example by filtering out parts of the result. As an example, the query in Algorithm 4.6 selects all friends of John Doe that have at least ten outgoing relationships [3]. It includes some new concepts, starting with a directed edge where the direction is given by an arrow ("<-[]-" or "-[]->"). Note however that the data model of Cypher does not support undirected edges at all, meaning that we cannot create or store undirected edges. The pattern -[]- that we used in the queries until now allows us to test for the existence of some directed relationship, regardless of the direction. Furthermore, there are two anonymous edges and one anonymous node in this query. These are entities that have no

[3] Note again that the edge label KNOWS depicts a friendship relation in the benchmark.

variable in their corresponding brackets and can therefore not be accessed later on in the query or included in the output. In this scenario however, we are not interested in the edges that match the directed edge-pattern, but only in the amount of such edges. Cypher also provides a short form for such anonymous edges: "- -" for edges where we do not specify the direction as well as "<- -" and "- ->" for directed ones. The `WITH` clause is needed in this query as we want to further limit our selection to friends that have at least 10 such edges. To this end, we group by the friends `f` and call the aggregate operation `COUNT` in the `WITH` clause that in turn forwards the resulting table to the `WHERE` clause where we can use the previously computed values.

```
1 MATCH (p:Person {firstName: 'John', lastName: 'Doe'}) -[:KNOWS]- (f:Person)
      -[]-> ()
2 WITH f, COUNT(*) AS out
3 WHERE out >= 10
4 RETURN f.firstName, f.lastName
```

Listing 4.6: A Cypher query that selects all friends of John Doe that have at least ten outgoing relationships.

Now that we have seen some of the general principles of Cypher, we will go over the queries from Chapter 3 and highlight advantages and disadvantages compared to other graph query languages. We took most implementations of the benchmark queries from their reference implementations on GitHub[4]. To ensure the correctness of the queries, we executed all of them on a Neo4j Community Edition 4.1.0 database. However, unless stated otherwise, the queries use only these functions and procedures that are also available in Cypher 9.

4.1.1 Structure Independent

We have already seen how we can formulate Query 1: either by restricting the matched nodes in the `WHERE` clause as in Algorithm 4.1 or by using the short form as in Algorithm 4.2. Cypher also supports the parameterization of queries such that the user can pass arguments to the query. An argument is then accessible by preceding its name with a dollar sign as we can see in Algorithm 4.7.

```
1 MATCH (p:Person {firstName:$firstName, lastName:$lastName})
2 RETURN p
```

Listing 4.7: Query 1 as a Cypher query.

The implementation of Query 2 in Cypher is given in Algorithm 4.8. As mentioned in the introduction of the query in Chapter 3, it returns the creation date of a given message and either the textual content in case there is one, or the image file (that is also saved as string). This differentiation can be seen in lines 4-6 of the algorithm where a `CASE` expression distinguishes between the two cases. To determine whether a message has the

[4] https://github.com/ldbc/ldbc_snb_implementations

property `content`, Cypher provides the `EXISTS` function. This function can be used on nodes and edges to test for the existence of properties.

```
1 MATCH (m:Message {id:$messageId})
2 RETURN
3   m.creationDate AS messageCreationDate,
4   CASE EXISTS(m.content)
5     WHEN true THEN m.content
6     ELSE m.imageFile
7   END AS messageContent
```

Listing 4.8: Query 2 (IS4) as a Cypher query.

Before we take a look at the implementation of Query 3, we have to look at Cypher's capabilities regarding collections such as lists. The benchmark states that the languages spoken by a person are stored in a list of strings and Cypher indeed supports lists as well as maps. Lists can be created simply by surrounding elements of the same type by square brackets as in `[0,2,4]` or by using the `collect` aggregation. Algorithm 4.9 depicts a Cypher implementation of Query 3. After selecting all persons in line 1, we use `SIZE` to get the number of entries in the `speaks` property of an individual `Person` (which is a list of strings). Thereafter, Cypher implicitly groups all of these values and calculates the average over the sizes. Apart from the aggregate operation `AVG`, Cypher also supports `MAX`, `MIN`, `COUNT`, `SUM` as well as functions to calculate the standard deviation and percentiles.

```
1 MATCH (p:Person)
2 RETURN AVG( SIZE( p.speaks ) )
```

Listing 4.9: Query 3 as a Cypher query.

We have already mentioned the `collect` function and we will now go briefly over the query in Algorithm 4.10 to give an example of it. The query selects all entities (edges and nodes) that are directly related to a continent whose name starts with `'A'`. At the end of the first query part we collect the names of these continents into the list `conts` that is passed to the `UNWIND` clause. There, the list is dissected again and each element in the list is passed on as `cont` to the remaining query part, meaning that this results in one execution of the latter `MATCH` for each entry in the list. The matches from these executions are then implicitly brought together and we can access all of them in the `RETURN` clause. Note that there are simpler ways to specify this query and we only use this as it allows us to show both, `collect` and `UNWIND` in one query.

```
1 MATCH (c:Continent)
2 WHERE c.name STARTS WITH 'A'
3 WITH collect(c.name) as conts
4 UNWIND conts AS cont
5 MATCH (c:Continent {name: cont}) -[r]- (n)
6 RETURN c, r, n
```

Listing 4.10: A Cypher query that selects all entities that are directly related to a continent that starts with 'A'.

4.1.2 Pattern Matching

We have already seen that patterns are used as main building blocks in Cypher queries. Caused by the fact that we have already given multiple examples of queries that contain simple bgps, we skip the code of Query 4 and continue with the similar looking Query 5. Algorithm 4.11 shows an implementation of this query. It includes an `ORDER BY` clause that works as expected for developers coming from SQL, meaning that the results are first ordered by the `friendshipCreationDate` in descending order and, on identical values on that variable, on the `personId` in ascending order.

```
1 MATCH (n:Person {id:$personId}) -[r:KNOWS]- (friend)
2 RETURN
3   friend.id AS personId, friend.firstName, friend.lastName,
4   r.creationDate AS friendshipCreationDate
5 ORDER BY friendshipCreationDate DESC, toInteger(personId) ASC
```

Listing 4.11: Query 5 (IS3) as a Cypher query.

To understand the implementation of Query 6 in Algorithm 4.12, we first have to introduce the `OPTIONAL MATCH` clause. Similarly to `MATCH`, we can specify a pattern in this clause and the database system searches for matches in the data. However, if there is no such match or only a partial one, meaning that for example only some variables of the `OPTIONAL MATCH` clause can be matched, all unmatched ones are set to `null`. This resembles the semantics of outer join in SQL. In our example, we first search for a pattern between a given message `m`, a replying comment `c` and its creator `p`. To test whether `p` knows the person that created the original message `m`, we optionally match the path from `m` to its creator and furthermore over a `KNOWS` edge to `p`. However, we do not know if the `KNOWS` edge exists as this is exactly what we want to find out. Therefore, we can then use a `CASE` expression in the `RETURN` clause to test whether such a match was found.

```
1  MATCH (m:Message {id:$messageId}) <-[:REPLY_OF]- (c:Comment)
2    -[:HAS_CREATOR]-> (p:Person)
3  OPTIONAL MATCH (m) -[:HAS_CREATOR]-> (a:Person) -[r:KNOWS]- (p)
4  RETURN
5    c.id, c.content, c.creationDate AS commentCreationDate,
6    p.id AS replyAuthorId, p.firstName, p.lastName,
7    CASE r
8      WHEN null THEN false
9      ELSE true
10   END AS replyAuthorKnowsOriginalMessageAuthor
11 ORDER BY commentCreationDate DESC, replyAuthorId
```

Listing 4.12: Query 6 (IS7) as a Cypher query.

Evaluation Semantics

Regarding the evaluation semantics of a bgp, Cypher uses *relationship isomorphism* as a default. That resembles our no-repeated-edge semantics where no two edge variables can be matched to the same edge in a single match. This avoids a potential infinite result in patterns with variable length [AAB+18]. It is envisioned that future versions of

Cypher, i.e. Cypher 10 under the openCypher project, allow the user to switch between a homomorphism-based, a no-repeated-node and the current no-repeated-edge semantics on a query-by-query base. [GJK+18]

However, we can influence the semantics of a query by our choice of the query structure even today. As an example, assume that we are searching for all friends-of-friends (of length exactly 2) of a given person `n`. We can then collect all these friends in a list and test whether `n` is in that list or not. The first example in Algorithm 4.13 contains a single `MATCH` clause and, caused by the no-repeated-edge semantics, returns `false` as the same edge will not be matched to both edges in the pattern. However, the version in Algorithm 4.14 returns true as there are no two edges in a single `MATCH` but they are split up into multiple ones.

```
1 MATCH (n:Person {id:$id}) -[:KNOWS]- (friend) -[:KNOWS]- (foaf)
2 WITH n, collect(foaf) as foafs
3 RETURN n IN foafs     //returns false
```

Listing 4.13: A cypher query that returns `false` as the same edge will not be matched to both edges in the pattern.

```
1 MATCH (n:Person {id:$id}) -[:KNOWS]- (friend)
2 MATCH (friend) -[:KNOWS]- (foaf)
3 WITH n, collect(foaf) as foafs
4 RETURN n IN foafs     //returns true
```

Listing 4.14: A cypher query that returns `true` as the same edge will be matched to the edges in both `MATCH` clauses.

Apart from that, Cypher also provides `UNION` and `UNION ALL` clauses that allow for a combination of results from multiple queries. Regarding the evaluation semantics of such queries, Cypher uses a bag semantics which is not problematic as we know that the evaluation of a single query results in a finite number of matches. [AAB+17, ARV19]

4.1.3 Path

Cypher supports a subset of RPQs as it allows for variable length paths. These paths can be further restricted by a single relationship type or a disjunction of such [FGG+18b]. In the most general form, we can abbreviate an otherwise unrestricted path with three edges and four nodes by `(a) -[*3]- (b)`. Furthermore, we can specify an upper bound `y` on the length via `[*..y]`, a lower bound `x` via `[*x..]` and a combination of both via `[*x..y]`. This can be extended to also include edge labels. As an example, we can search for the friends-of-friends of length 3 via `(a)-[:KNOWS*3]-(p)`. Besides paths of variable length, a disjunction of labels is given as `(a)-[:KNOWS|WORK_AT]-(b)`. However, Cypher does not support full composition of regular expressions over edge labels and its path querying functionality is therefore not as expressive as RPQs.

Apart from restrictions on paths, Cypher also allows paths to be assigned to variables. As an example, we can assign a path to a variable as in Algorithm 4.15. This allows

us to further restrict a path, for example in a `WHERE` clause, and to return it. Cypher provides functions like `nodes` and `relationships` to access these entities on a path in order to restrict or return them as well as `length` to get the length of a path.

```
1 MATCH p = (n:Person {id:$id}) -[:KNOWS*..5]-> ()
2 WHERE length(p) > 2
3 RETURN relationships(p), length(p) LIMIT 10
```

Listing 4.15: A Cypher query that demonstrates named paths (paths that are assigned to variables).

Algorithm 4.16 depicts an implementation of the reachability query 7. This is an interesting query as it searches for the shortest path between two nodes with a restriction on the edges that can be traversed. Cypher provides the `shortestPath` function for exactly that, meaning that the function finds "*a single shortest path between two nodes*" [ope18, Neo20]. However, it is not specified which path is returned when there are multiple shortest ones. As we do not know if such a path exists at all in our case, we use the function in an `OPTIONAL MATCH` so that we can later test whether such a path was found or not. Apart from the `shortestPath` function that returns a single shortest path, Cypher also provides the `allShortestPaths` function that returns all shortest paths between two nodes.

```
1 MATCH (person1:Person {id:$person1Id}), (person2:Person {id:$person2Id})
2 OPTIONAL MATCH path = shortestPath((person1)-[:KNOWS*]-(person2))
3 RETURN
4   CASE path IS NULL
5     WHEN true THEN -1
6     ELSE length(path)
7   END AS shortestPathLength;
```

Listing 4.16: Query 7 as a Cypher query.

Algorithm 4.17 contains an implementation of the adjacency query 8. The query incorporates multiple concepts and clauses that we explained before and at a first glance looks quite complicated. However, this comes mostly from the fact that it starts with a simple pattern that gathers friends up to distance 3 and then proceeds by collecting further information about them in multiple steps. As there can be more than one path of length 1 to 3 between two persons that know each other, we use the `shortestPath` function to get a single one. After excluding the start person in line 2, we pass the required information to the next query part via the `WITH` clause. This part is then evaluated once for every friend from the first part and gathers information about the place the friend is living at and optionally the place(s) he or she studied at. Information about a single university is gathered in a manually created list in line 12. As a person can have relations to none, a single or multiple universities, we iterate over them and collect the information in a list using `collect`. The information is combined in another `WITH` clause, forwarded to the next query part where information about the work-relationships of the friend are added and finally returned. This query shows how we can iteratively

gather information and that the information we want to finally return has to be passed to the end via `WITH` clauses.

```
1  MATCH p=shortestPath((person:Person {id:$personId}) -[path:KNOWS*1..3]- (
       friend:Person {firstName:$firstName}))
2  WHERE person <> friend
3  WITH friend, length(p) AS distance
4    ORDER BY distance ASC, friend.lastName ASC, toInteger(friend.id) ASC
5  MATCH (friend) -[:IS_LOCATED_IN]-> (friendCity:Place)
6  OPTIONAL MATCH (friend) -[studyAt:STUDY_AT]-> (uni:Organisation) -[:
       IS_LOCATED_IN]-> (uniCity:Place)
7  WITH
8    friend,
9    collect(
10     CASE uni.name
11       WHEN null THEN null
12       ELSE [uni.name, studyAt.classYear, uniCity.name]
13   END
14   ) AS unis,
15   friendCity,
16   distance
17 OPTIONAL MATCH (friend) -[workAt:WORK_AT]-> (company:Organisation) -[:
       IS_LOCATED_IN]-> (companyCountry:Place)
18 WITH
19   friend,
20   collect(
21     CASE company.name
22       WHEN null THEN null
23       ELSE [company.name, workAt.workFrom, companyCountry.name]
24     END
25   ) AS companies,
26   unis,
27   friendCity,
28   distance
29 RETURN friend, distance, friendCity.name, unis, companies
```

Listing 4.17: Query 8 as a Cypher query.

We skip the implementation of Query 9 as it does not provide any new insights and can be written as a relatively simple navigational query.

Evaluation Semantics

Similar to the evaluation semantics of pattern matching queries, Cypher uses a no-repeated-edge semantics when evaluating path queries [DXWL19]. Therefore, the evaluation of such a query will never take the same edge in a path, which in turn avoids paths of unlimited length. Regarding possible outputs of a query, Cypher supports everything from fixed arity outputs like Booleans, single properties and entities to variable arity outputs like paths and lists. However, we are not able to return a graph in Cypher as of now, although something in that regard should be supported in Cypher 10 [FGG+18b].

4.1.4 Data Manipulation

Now that we have seen a variety of queries that select data from the graph, we will look at DML queries. As mentioned in Chapter 3, queries that manipulate data are expressed similarly to ones that only retrieve data. This can be seen in the simplified implementation of Query 10 in Algorithm 4.18 that uses the same patterns also to create nodes and relationships. The query inserts a new person node via the `CREATE` clause as well as a relationship labeled `IS_LOCATED_IN` to connect the new node to the city the person is living in. All of that is done in the first two lines and the code in the following lines then creates a variable number of relationships to already existing nodes. We use `UNWIND` to unroll the list of `tags`, `study` and `work` information and each such entry is then passed to one `MATCH-CREATE` combination. As an interesting side note, we have to include `count(*)` in the `WITH` clauses to avoid an unexpected behavior: if we pass for example two tags in `tagIds`, this results in two evaluations of the `MATCH-CREATE` combination in lines 5 and 6 which in turn would result in two calls of the `WITH` clause in line 7. By adding `count(*)` in that clause, we implicitly group on the single key `p` which in turn is the only value that is then passed to the next query part.

```
1  MATCH (c:City {id:$cityId})
2  CREATE (p:Person {id: $personId, firstName:$fn, birthday: date($bd),
       creationDate: timestamp($cd), emails: $emails}) -[:IS_LOCATED_IN]-> (c)
3  WITH p
4  UNWIND $tagIds AS tagId
5      MATCH (t:Tag {id: tagId})
6      CREATE (p) -[:HAS_INTEREST]-> (t)
7  WITH p, count(*) AS dummy1
8  UNWIND $studyAt AS s
9      MATCH (u:Organisation {id: s[0]})
10     CREATE (p) -[:STUDY_AT {classYear: s[1]}]-> (u)
11 WITH p, count(*) AS dummy2
12 UNWIND $workAt AS w
13     MATCH (comp:Organisation {id: w[0]})
14     CREATE (p) -[:WORKS_AT {workFrom: w[1]}]-> (comp)
```

Listing 4.18: Simplified version of Query 10 (INS1) in Cypher that inserts only some properties.

Apart from `CREATE`, Cypher also provides the `MERGE` clause that ensures that a given pattern exists in a graph. If the pattern already exists, it works similar to `MATCH` as the variables are bound to the existing and matched entities. Otherwise, the entities are first created and then bound to the variables.

Cypher provides two clauses to remove entities from the graph: `DELETE` removes a single node or edge whereas `DETACH DELETE` removes a node together with all incident edges. Regarding the removal of connected nodes from the graph, stale edges are not allowed at any time. Therefore, a query that deletes a node without also removing its relationships will fail. That also means that the removal of a connected node will fail via the `DELETE` command. Using `DETACH DELETE` however, we are able to remove connected nodes and

trigger recursive removals via a short query. This can be seen in the implementation of Query 11 in Algorithm 4.19.

```
1 MATCH (comment:Comment {id: $commentId}) <-[:REPLY_OF*]- (replies:Comment)
2 DETACH DELETE comment, replies
```

Listing 4.19: Query 11 (DEL7) as a Cypher query.

Apart from inserting and deleting nodes and edges, we can also update their properties, add new ones or remove existing ones. An example of such an update can be seen in Algorithm 4.20 where we change an existing property. The `SET` command is also used to insert a new property, meaning that Cypher does not test whether the property `classYear` already exists but simply saves it with the new value, overwriting a potentially existing one or creating it anew. To remove a property from a node or an edge, Cypher provides the `REMOVE` clause. Nodes in Cypher's data model can have multiple labels where the same commands are used to update, add or remove them. Edges however can only have one label that is set at edge creation and cannot be updated or removed later on.

```
1 MATCH (:Person {id:$personId}) -[r:STUDY_AT]-> (u:University {id:$univId})
2 SET r.classYear=2020
```

Listing 4.20: Query 12 that sets (updates or adds) the property `classYear`.

4.1.5 Data Definition

Cypher in its original version does not support a schema and "*was originally conceived in a dynamically typed, schema-less context*" [FGG+18b] that works on a single graph only. That is still true in the current version governed by openCypher [ope18], but other implementations like *Morpheus: Cypher for Apache Spark*[5], *SAP HANA Graph*[6] and Cypher in Neo4j support at least some kinds of (schema) constraints and more than one graph. We will now look at the constraints available in Neo4j as these will likely also be available in future versions of openCypher as there exists a Cypher Improvement Proposal[7] for them in openCypher. *Unique node property constraints* ensure that the values of a property on a node with a specific label are unique. This constraint however does not enforce that the property has to exist on all nodes with this label but only that if the property exists, its value is unique. *Node property existence constraints* allow us to specify that all nodes with a specific label have to have a specific property and *relationship property existence constraints* are the counterpart for edges. Finally, *node key constraints* ensure that a set of properties has to exist on each node with a given label and the combination of the property values has to be unique. We could enforce at least parts of a schema using the latter three constraints. However, they are only available in the Enterprise Edition of the Neo4j database whereas the open source Community Edition is limited to unique node property constraints.

[5] https://github.com/opencypher/morpheus
[6] https://www.sap.com/products/hana.html
[7] https://github.com/opencypher/openCypher/pull/166

Even if we were to use Cypher as in Neo4j's Enterprise Edition, we could not implement Query 13 as there is no way to specify allowed patterns. The only thing we can do is to change the existing relationships in a way that they comply with the the query. To cope with the lack of a schema in general, Cypher provides functions like `EXISTS`, that can also test for the existence of patterns, or the `OPTIONAL MATCH` clause. However, having a schema would allow a system to fully type check queries up-front and add additional optimization potential for the query planner and the execution [FGG+18b].

4.1.6 Further Features

- **Composability:** We have already seen that Cypher supports linear composition of queries via the `WITH` clause. If we look into (nested) subqueries, we are again limited to versions of Cypher different from openCypher. There is however a Cypher Improvement Proposal[8] that targets version 10 of the language and introduces this feature in the openCypher version. We will now look at an example of a correlated subquery as it is currently supported by the version of Cypher in Neo4j. As an example, assume that we want to calculate the number of persons that are older than John Doe. Algorithm 4.21 depicts an implementation of this query where we use the "`CALL {}`" clause[9] to evaluate the subquery. Again, we have to use `WITH` to specify which elements of the binding table from the outer query are passed to the subquery.

```
1 MATCH (p:Person {firstName: 'John', lastName: 'Doe'})
2 CALL {
3   WITH p
4   MATCH (o:Person)
5   WHERE o.birthday < p.birthday
6   RETURN COUNT(o) as youngerCount
7 }
8 RETURN p.firstName, p.lastName, youngerCount
```

Listing 4.21: A Cypher query in Neo4j's version that selects John Doe and counts the number of older persons.

However, Cypher allows us to formulate many such queries in a way that avoids subqueries altogether. The query in Algorithm 4.22 is equivalent to the previous one but achieves this via two (uncorrelated) matches that are then related to each other in the `WHERE` clause. We use `WITH` in this implementation to group by p and count the matches of younger persons before we return them.

```
1 MATCH (p:Person {firstName: 'John', lastName: 'Doe'})
2 MATCH (o:Person)
3 WHERE o.birthday < p.birthday
```

[8] https://github.com/opencypher/openCypher/pull/100

[9] Note that there is also a "`CALL`" clause available in both, Neo4j's and openCypher's version, that invokes a procedure. The "`CALL {}`" clause that invokes a subquery however is only available in Neo4j's version.

```
4 WITH p, COUNT(o) as youngerCount
5 RETURN p.firstName, p.lastName, youngerCount
```

Listing 4.22: An alternative version of the query from Algorithm 4.21 that avoids the subquery.

Neo4j's version of Cypher furthermore allows for existential subqueries using a combination of `WHERE` and `EXISTS`. Unlike explained above, queries nested through `WHERE EXISTS` can access variables from the outer scope even without `WITH`. On the other hand, such queries can often be reformulated in a way that they avoid existential subqueries altogether by using a more sophisticated pattern, `OPTIONAL MATCH` or specifying the requirements in the `WHERE` clause.

Apart from that, Cypher in its current state is not a composable language as a query outputs a table but takes a graph and an (empty) table as input. There is however again a Cipher Improvement Proposal[10] to include this in Cypher 10 via a change of the binding tables. The plan is to adapt these tables so that they not only include tabular information but also one or multiple graphs, therefore becoming *table-graphs* [FGG+18b]. This would enable query composability but also requires a change of other structures as Cypher 9 cannot deal with multiple graphs. There is already one implementation of Cypher that supports these features, namely Morpheus, the version for Apache Spark.

- **Paths as first-class citizens:**
  We mentioned before that Cypher allows paths to be matched and returned. However, this is not enough for it to treat them as first-class citizens as we cannot save paths or add properties to them and to the best of our knowledge, there is no attempt to change this as of now.

As we have seen in this section, Cypher focuses heavily on patterns and provides a syntax that allows for compact queries in many cases. It provides a myriad of mathematical and string functions that we did not introduce as well as aggregating and graph-specific functions like `shortestPath, nodes, labels`. The version of Cypher in Neo4j also includes temporal functions for dates, times including time zones and durations as well as spatial functions. Apart from these functions that are included in the language, we can extend Cypher via user-defined functions. These functions are deployed into the database and can then be called similar to Cypher functions in queries. The Neo4j database also supports user-defined procedures that can take arguments and interact with the database before returning a result. In Neo4j, these functions and procedures are written in Java and the company provides some of them in bundles. Awesome Procedures On Cypher (APOC)[11] and a bundle especially for data scientists[12] are examples for them.

[10] https://github.com/opencypher/openCypher/pull/241

[11] https://neo4j.com/labs/apoc/, https://github.com/neo4j-contrib/neo4j-apoc-procedures

[12] https://github.com/neo4j/graph-data-science/

4.2 Gremlin

Unlike Cypher that is a high-level declarative language, the Gremlin query language [Tin20] is a low-level language that offers imperative as well as declarative constructs [Rod15]. Although it is a functional language at its core, it is more imperative in nature and focuses on graph traversals instead of pattern matching [AAB+17]. However, apart from its focus on traversals and therefore navigational queries in the imperative style, Gremlin provides pattern matching features in a declarative construct [TPAV19]. From the earliest prototypes onwards, Gremlin extensively uses XPath to provide complex graph traversals [Lin18, AAB+18].

As briefly mentioned in the introduction of Gremlin in Chapter 2, Gremlin denotes not only the query language but also the traversal machine GTM used in the TinkerPop framework. A query written in the Gremlin query language is compiled to generate a *traversal* that is then executed on the GTM. In the imperative traversal mode, that is also used in some sense to match patterns as we will see later on, the user specifies so called *motifs*, that are traversal instructions containing multiple *steps*. These steps are then executed by a *traverser*, an instance of a *traversal*, in the GTM. In other words, a single traversal has a set of traversers attached to it that walk on a graph according to the steps from the query. The result of such a traversal then contains all locations where a traverser halted. From now on, we mean the query language and not the traversal machine when we speak about Gremlin. [Rod15, AAB+17]

Before we can examine our first query on the third line in Algorithm 4.23, we have to create a traversal source on a graph. Such a source is needed to create a traversal that in turn resembles a query and interacts with a graph. After connecting to some graph and assigning it to the `graph` variable in line 1, we create the traversal source `g` in line 2. This traversal source can be used not only for a single query but for all queries on the `graph`. The query in line 3 selects all properties (`values`) of John Doe. Similar to the linear structure of Cypher queries, a query in Gremlin is always evaluated from left to right. Starting from the traversal source `g`, we first generate a traversal that starts at the vertices in the graph via `V()`. Intuitively, there is now one traverser on each vertex in the graph and the following steps specify how they proceed. If the query was to end here, meaning that it only contains `g.V()`, we get all nodes as a result since there is one traverser on each node after the execution of `V()`. In our query in Algorithm 4.23 however, we continue with a `has()` step to remove all traversers where the corresponding element (a node in our case) does not have the given key/value property. Therefore, we are left with traversers on only those nodes that have a 'firstName'/'John' property. Note that this also removes all traversers on nodes that do not resemble a person as these do not have a `firstName` property. After repeating this step to also remove traversers on nodes that do not have the 'lastName'/'Doe' property, we select all property values of the node with `values`.

Note that the commands can be run as given in the algorithm in the *Gremlin Console*, an interactive terminal that connects to the GTM and allows for ad hoc queries to be

executed. For queries in applications however, one normally uses an implementation of Gremlin in a programming language and interacts with a GTM from there. We will go into more details regarding such implementations and their differences detail later on.

```
1 gremlin> graph = TitanFactory.open(<graph>)
2 gremlin> g = graph.traversal()
3 gremlin> g.V().has('firstName','John').has('lastName','Doe').values()
```

Listing 4.23: An example on how to connect to a graph, create a graph traversal source on the graph and spawn a traversal in the Gremlin Console. The query then selects the properties of node(s) with the name `John Doe`.

The query in Algorithm 4.24 selects the names of all friends of John Doe. In the previous query, we removed all traversers on nodes that do not resemble a person implicitly by requiring the property `firstName`. We can also state such a label requirement explicitly via the `hasLabel()` step. After the steps in line 1 are executed, we are left with a traverser on the node representing John Doe (assuming that there is only one such node). The next step, `out()`, is a vertex step that allows us to traverse over an outgoing edge to an adjacent vertex, denoted as outgoing adjacent in Gremlin. In our query, we limit the traversal to `knows` edges (edges labeled with "knows"). In case there are multiple such adjacent vertices, this step essentially spawns new traversers and assigns each of those vertices one traverser. Apart from moving to outgoing adjacent vertices, Gremlin also supports `in()` to move to incoming adjacent vertices (adjacent vertices connected over incoming edges) and `both()` to move to both, incoming and outgoing adjacent ones. In the last line of this query, we select the first and last name of the friends of John Doe.

```
1 g.V().hasLabel('person').has('firstName','John').has('lastName','Doe')
2   .out('knows')
3   .values('firstName', 'lastName')
```

Listing 4.24: A Gremlin query that selects the names of John Doe's friends.

As we have seen in this query, we are forced to think in a graph-perspective when writing a query in Gremlin [Rod15]. Until now, this perspective is mostly limited to vertices as we can select and filter them and move from one to another. However, we can also move to edges which can be seen in Algorithm 4.25. This query selects a map containing all key/value pairs of outgoing `knows` edges of John Doe via `valueMap()`. The `outE()` step is again a vertex step but unlike the ones mentioned before, moves to outgoing incident edges. Similar to the vertex steps from above, Gremlin also provides steps like `inE()` to move to incoming incident edges and `bothE()` to move to both, incoming and outgoing ones. All steps mentioned until now can be invoked on traversers on vertices. Furthermore, steps like `outV()`, `inV()`, `bothV()` and `otherV()` can be used on traversers on edges that allow us to move from an edge to a vertex.

```
1 g.V().has('firstName','John').has('lastName','Doe').outE('knows').valueMap()
```

Listing 4.25: A Gremlin query that selects properties of outgoing `knows` edges of John Doe.

All of our queries until now are composed as a concatenation of steps which looks similar to a general programming language. And indeed, Gremlin is sometimes denoted as a "*programming language for property graphs*" [AAB+18]. This sets it apart from most other query languages, especially also Cypher, and allows us for example to store the result from one query in a variable and use it later on in a different query. Algorithm 4.26 contains some examples for this where each line depicts a single query. The query in the first line assigns the node depicting John Doe to the variable `john`. In order to assign it, we have to use the `next()` step that triggers the evaluation. This is needed as Gremlin uses a lazy evaluation on most queries. We were able to omit this step in the previous queries as queries entered in the Gremlin Console are evaluated automatically after pressing Enter. However, that does not work when assigning a query result to variables, where we have to manually trigger the evaluation of the query to assign the result. Now that we have a reference to this node, we can use this variable in the following queries. As an example, we directly start from the node `john` in the query in line 3 and traverse to vertices connected over `knows` edges and select their `firstName`.

Another area where Gremlin differs from most query languages is the support for lambdas. Lambda steps like `map(), flatMap()` and `filter()` allow for custom functions that can be used to restructure or filter data. Although these steps "*represent the foundational constructs of the Gremlin language*" [Tin15, Tin20], they should be avoided if possible as they cannot be inspected and optimized by the compiler. Nonetheless, having the ability to use lambdas in case there is no lambda-less step directly provided by Gremlin allows a user to essentially run any function. Therefore, it is also not surprising that Gremlin is a Turing-complete language [WD18]. The queries in lines 3, 4 and 5 are equivalent and while the first version uses Gremlin functions, the latter ones use the lambda step `map()`.

```
1 gremlin> john = g.V().has('firstName','John').has('lastName','Doe').next()
2          //the following three queries are equivalent:
3 gremlin> g.V(john).out('knows').values('firstName')
4 gremlin> g.V(john).out('knows').map(values('firstName'))
5 gremlin> g.V(john).out('knows').map {it.get().value('firstName')}
```

Listing 4.26: Multiple Gremlin queries that assign and use a variable and demonstrate the use of lambda functions.

Each query until now started with the `V()` step, that spawns a traversal on the vertices from the traversal source `g`. Gremlin also allows a traversal that starts on the edges via the `E()` step as can be seen in Algorithm 4.27. `groupCount()` is a step that counts how often an object has been part of a traversal and can therefore not only summarize labels but also any other information like the names or ages of persons.

```
1 g.E().label().groupCount()
```

Listing 4.27: A Gremlin traversal starting from the edges that generates a summary of the edge labels.

We can furthermore use `union()` to combine the results of multiple traversals or query parts. As we can see in Algorithm 4.28, we can save a reference to an object using `as(<ref>)` and can later on in the query access it via `select(<ref>)`. This query selects a person with a given id[13] and combines the first name of the person with the number of outgoing `knows` edges in the `union()` step. The last step, `fold()`, aggregates these two values into a list and emits it.

```
1 g.V().has('iid','person:933').as('person').
2   union(
3     select('person').values('firstName'),
4     outE('knows').count()
5   ).fold()
```

Listing 4.28: A gremlin query that outputs information about a specific person and counts the number of outgoing edges.

All queries in this chapter were tested on a Titan 1.0.0 database[14] that natively supports the TinkerPop graph stack and therefore also the Gremlin query language. However, as Titan is no longer maintained (but a fork of it, JanusGraph[15], is), it does not use the most recent version of TinkerPop and therefore Gremlin [Tin20] but the older version 3.0.1-incubating [Tin15]. As we ran all queries on Titan, they are compatible with version 3.0.1 and do not use additions that were introduced in later versions. An example of such an addition is the `propertyMap()` step that extends `valuesMap()` to also include the id and the label of an entity. However, all of the core functionalities and steps are also available in version 3.0.1.

Apart from the database we will briefly explain the workings of the Gremlin Console, why there are implementations of Gremlin in multiple programming languages and which one we use in our queries. The TinkerPop project provides a reference implementation of Gremlin in Java 8: *Gremlin-Java*. However, users are free to implement a variant of Gremlin in any other programming language as long as the language supports function composition and function nesting. Such an implementation provides the step functions and is responsible for the translation of queries to Gremlin Bytecode that in turn can be executed on a GTM. Apart from the implementation in Java, TinkerPop also provides an implementation in Apache Groovy: *Gremlin-Groovy*. This language variant is used in the Gremlin Console. Therefore, the Gremlin queries we have seen until now can all be run on every Groovy-system that includes the required libraries for Gremlin. However, the only notable differences between these two language variants regards the use of single versus double quotes for strings (Groovy allows single quotes, Java requires double quotes), and a small change in syntax on anonymous entities. The following queries are implemented in Gremlin-Java. We pooled the benchmark queries from two GitHub repositories: the

[13] We use the property *iid* in our data model for our custom ids as `id()` is a native Gremlin function that accesses the id generated by Gremlin and not the one generated by the LDBC data generator.

[14] `http://titan.thinkaurelius.com/`

[15] `https://janusgraph.org/`

original implementation from PlatformLab[16] and a fork and extension of this from Anil Pacaci[17].

4.2.1 Structure Independent

We have already seen how a person node with a given first and last name can be selected, which is exactly what the query in Algorithm 4.29 does. Nonetheless, we give this as an example of a query written in Gremlin-Java, where the only difference to queries from before is the usage of double quotes for strings That means, the query can be executed directly in any Java application that includes the Gremlin-Java libraries. Therefore, the usage of Gremlin differs from the use of query languages like SQL in application code as SQL code is usually provided as a string that is passed over an interface to the database. In contrast, a Gremlin query is implemented in the programming language itself and compiled to Bytecode to be executed. The query also shows an extension of the `has()` step that can require a node to have a label (the first argument) and a key/value property in a single statement.

```
1 String firstname = "John";
2 String lastname = "Doe";
3 System.out.println(g.V().has("person", "firstName", firstname).has("lastName"
    , lastname).valueMap());
```

Listing 4.29: Query 1 as a Gremlin query in the Gremlin-Java language variant.

Another example where we can see the interaction between code written in Java and the Gremlin query is given in Algorithm 4.30. Whereas the query itself is given in lines 1-3, the result of the query is accessed in the following lines where we check whether the post contains an `imageFile` or a textual `content`. Nonetheless, we run into a limitation of Gremlin when implementing this query as the data model only supports a single label on vertices. In our case however, `posts` and `comments` are both `messages` and we labeled them either with "`post, message`" or "`comment, message`" when working with Cypher. Since the corresponding nodes in Gremlin can only have a single label, they are either labeled by "`post`" or "`comment`", and we cannot directly search for a `message`. We achieve this functionality in Gremlin using the `within` predicate in a `has()` step to filter nodes labeled with `post` or `comment`. Apart from that, the algorithm shows that accessing the results from Java is not as straightforward as one might think since we need to repeatedly cast variables of type `Object` to `List<String>` in order to get the first entry of these lists. On a side note, the `T` and `P` prefixes are only needed in Gremlin-Java as a restriction from Java and can for example be omitted in Gremlin-Groovy.

```
1 Map<String, Object> values = g.V().has(T.label, P.within("post", "comment")).
2   has("iid", messageID).
3   valueMap("iid", "creationDate", "content", "imageFile").next();
4 List<String> result = new LinkedList<>();
```

[16] https://github.com/PlatformLab/ldbc-snb-impls
[17] https://github.com/anilpacaci/graph-benchmarking

```
5 result.add(((List<String>) values.get("iid")).get(0));
6 result.add(((List<String>) values.get("creationDate")).get(0));
7 if (((List<String>) values.get("content")).get(0).equals(""))
8   result.add(((List<String>) values.get("imageFile")).get(0));
9 else
10  result.add(((List<String>) values.get("content")).get(0));
```

Listing 4.30: Query 2 (IS4) as a Gremlin query in Gremlin-Java. We can see how results of a query can be accessed in Java.

Algorithm 4.31 contains two different implementations of Query 3. For this query, we have to access the multi-value property `speaks` of a person. Gremlin's data model allows for such multi-value properties, meaning that a single property key on a vertex can have multiple values attached to it. The first version saves a reference to a traversal in Java and iterates manually over the traversal to trigger the lazy evaluation and access the results. In the second version, we use two queries that calculate the two sums and then calculate the average in Java. However, both versions compute the average in Java and not in the query directly. While Gremlin provides steps to group, count and compute the average, it is not clear how to formulate this with the access to a multi-value property in a single query.

```
1 //Version 1:
2 Map<String, Object> map;
3 GraphTraversal<Vertex, Map<String, Object>> tr = g.V().hasLabel("person").
      values("speaks");
4 while(tr.hasNext()) {
5   map = tr.next();
6   overall += ((List<Object>) map.get("speaks")).size();
7   entries++;
8 }
9 double result = overall * 1.0 / entries;
10 log.info("Result of Query 3 (avg. spoken languages): " + result);
11
12 //Version 2:
13 long sumLangs = g.V().hasLabel("person").properties("speaks").count().next();
14 long sumPersons = g.V().hasLabel("person").count().next();
15 double res = sumLangs * 1.0 / sumPersons;
```

Listing 4.31: Two versions of Query 3 in Gremlin.

The two queries in the second variant implicitly group in the `count()` step. Apart from this, Gremlin provides a `group()` step where the user can specify the key to group by. Algorithm 4.32 contains some examples of group-by in the setting of a simple social network where users are stored as vertices with a `name` and `year` property. The query in line 1 uses `groupCount()` together with a `by()` step that specifies the key to group by. `by()` is not an actual step as it has no meaning on its own and can only be used together with steps like `groupCount()`, `group()` or `select()` as we will see later on. The query in line 3 is equivalent to the one in the first line, but shows how we can manually trigger the `grouping` and specify the key in the following `by()` step. Since we want to count

the persons for each year (grouping key), we end this query with another `by(count())`. Taken together, we can conclude that a group-by operation that aggregates the values in the groups is given as `group().by(grouping_axis).by(reducing_step)`. Apart from this, the same construct can also be used to project instead of reduce via `group().by(grouping_axis).by(projection_axis)`. The query in line 6 provides an example of this as it groups by `year` and returns a list of person names for each year.

```
1 g.V().hasLabel("person").groupCount().by("year")
2 //is equivalent to:
3 g.V().hasLabel("person").group().by("year").by(count())
4
5 //this creates a map with the year being the key and the names the values:
6 g.V().hasLabel("person").group().by("year").by("name")
7
8 //by can also contain a traversal, groups by the number of incident edges:
9 g.V().group().by(bothE().count())
```

Listing 4.32: Grouping operations in Gremlin queries.

Gremlin allows us to not only group by a property or a label but also on the result of a traversal. To this end, we can have a traversal inside any of the two occurrences of `by()` in a grouping operation, where the result either specifies the key to group or the projection. An example of this is given in line 9 where we group all vertices by their number of incident edges, both incoming and outgoing. Furthermore, `by()` also accepts functions in the `reducing_step`. Similar to other query languages, Gremlin provides common aggregate functions like `min`, `max`, `mean`, `count` and `sum`. Together with lambda functions, we can essentially provide any function as a function to reduce-by (aggregate function), even if it is not provided by Gremlin out of the box.

4.2.2 Pattern Matching

All queries until now used imperative traversals that are at the focus of Gremlin as query language, especially for navigational queries. However, Gremlin also provides the declarative pattern matching step `match()` that can be used in any traversal. That allows a query to switch from imperative traversals to declarative pattern matching and back again in a single query and with the same traversal source [Rod15]. In contrast to other graph query languages like Cypher, Gremlin also uses traversals and therefore a navigational approach to encode the structure of a pattern [AAB+17]. The navigational approach however results in pattern matching queries that are often not as compact or easy to understand as in higher level, declarative languages like Cypher. Caused by the focus on navigational queries, some pattern matching queries can be written even without `match()` as we will see in the following two algorithms.

The first query in Algorithm 4.33 is such an example where we do not have to use the `match()` step to depict the pattern. This is especially caused by the fact that we are looking for a small pattern in this case as we are only interested in the connection between

a `person` and the `city` he/she is living in. As mentioned before, the `as()` step allows us to store a reference to an entity that we can later access. This is not limited to the pattern matching part. In this case, we select the person node with the given `iid` and store a reference under the name `person`. We then proceed by navigating over an outgoing edge labeled by `isLocatedIn` and store that node as `city`. Now that we have those references, we can `select` the corresponding entities and project to some values. To this end, we append one `by()` for each entry in `select` to specify the projection for the corresponding entity.

An implementation of the same query using the pattern matching step is given in lines 6-9. The query starts, similar to many navigational queries, from the traversal source by spawning a traversal on the vertices. We then switch to the pattern matching step via `match()`. After saving a reference to the start node of the traversal (`person`), we proceed as we would in a navigational query. Therefore, the implementation using pattern matching in this scenario looks very similar to the one using navigational constructs only. However, we will see a more sophisticated example later on in Algorithm 4.35. On a side note, the query selects all properties of the `person` and the `iid` of the city. That does not exactly match the definition of the query in the SNB. However, the version of Gremlin that we use (3.0.1-incubating) does not provide a step to project to certain properties and we will show a way to deal with that in the next query.

```
1 g.V().has("person", "iid", personId).as("person").
2   out("isLocatedIn").as("city").
3   select("person", "city").by(valueMap()).by("iid");
4
5 //equivalent version using match:
6 g.V().match(
7   __.as("person").has("person", "iid", personId).
8     out("isLocatedIn").as("city")
9 ).select("person", "city").by(valueMap()).by("iid")
```

Listing 4.33: Query 4 (IS1) as a Gremlin query.

As an implementation of Query 5 using the pattern matching constructs looks very similar to the one from the previous query, we will only look at the implementation using traversals in Algorithm 4.34. The traversal starts at the person with a given `iid` and then switches from that node to outgoing `knows` edges. Line 2 therefore operates entirely on those outgoing edges and includes the `order().by()` steps that order them according to the descending `creationDate`. We then move to the other vertex on the edge, the one that we did not came from, via the `otherV()` step. As the output should contain three properties of these vertices, we save three references to them. Finally, we select all these references and specify the projection-axis for each of them. This achieves the same functionality as the `project()` step that was introduced in newer versions of Gremlin.

```
1 g.V().has("person", "iid", personId).
2   outE("knows").order().by("creationDate", decr).as("knows").
3   otherV().as("a", "b", "c").
```

```
4   select("a", "b", "c", "knows").
5       by("iid").by("firstName").by("lastName").by("creationDate")
```

Listing 4.34: Query 5 (IS3) as a Gremlin query.

The implementation of Query 6 in Algorithm 4.35 is an example of a pattern matching query where we use the `match()` step in a more sophisticated way. We have to start the pattern matching construct by saving a reference to the start node `m`. With that reference, we proceed with a navigational query in the rest of line 2. The comma at the end of line 2 intuitively ends this traversal and a new one is started at the beginning of line 3. We can either create a new reference, or, as we do here, re-use an existing one. Therefore, we start again at the node that we saved in `m` and proceed with another navigational query. As soon as we have all references that we need, we leave the `match()` step, therefore switching again to the imperative traversal where we select the values that should be included in the output. In this case, we omit the `by()` steps such that a reference to the entities themselves will be returned. This is again not what the definition of the query in the benchmark states. However, the query also lacks the boolean value that depicts whether the creator of the original message (`cr`) and the creator of the replying comment (`replyAuthor`) know each other. This value could be generated by a simple check in the programming language whether `cr` is included in `friends`. When working with Gremlin, many applications aggregate and sort the required information after the extraction, meaning that the query often selects more information than needed and a data structure in the programming language is then populated with the desired information afterwards.

```
1 g.V().match(
2   __.as("m").hasLabel("post").has("iid", messageId).out("hasCreator").as("cr"
      ),
3   __.as("m").in("replyOf").hasLabel("comment").as("co"),
4   __.as("co").out("hasCreator").hasLabel("person").as("replyAuthor"),
5   __.as("replyAuthor").out("knows").hasLabel("person").fold().as("friends")
6 ).select("m", "co", "replyAuthor", "friends")
```

Listing 4.35: Query 6 (IS7) as a Gremlin query. Note however that we are not able to generate the boolean value depicting whether `cr` and `replyAuthor` know each other inside the query.

Now that we have seen examples of patterns in Gremlin, we will go over some general remarks and further capabilities regarding patterns. Generally, Gremlin supports bgps, filters (for example the `where()` and `has()` steps), projection to some extent but also cgps using steps like `union()` [AAB+17, AAB+18]. Apart from the union of two traversals we can also specify branches via the `choose()` step and even cyclic occurrences via `repeat()`. Since traversals are used inside the declarative matching step to encode the pattern, we can use all of those constructs also when describing patterns. That enables expressive patterns and the documentation even states that it allows for "*patterns of arbitrary complexity*" [Tin15]. Furthermore, as Gremlin uses function composition

and nesting, and `match` is nothing but a function, we can recursively nest `match()` steps. Taken together, we can say that patterns in Gremlin add a declarative flavor to the language which however relies heavily on imperative constructs like the traversals. Compared to other graph query languages, this approach together with the language being low-level and having a compact syntax results in queries that are often not easy to read [HP13].

Evaluation Semantics
In contrast to Cypher, Gremlin uses the homomorphism-based bag semantics when evaluating a pattern [AAB+17]. Therefore, the matches are not restricted which can result in infinite result sets. That is also one of the reasons why Gremlin uses a lazy evaluation on most queries. Users have to be aware of this when writing queries as the evaluation of an unrestricted query can be non-terminating, depending on the concrete implementation of the language-variant.

4.2.3 Path

As briefly mentioned before, Gremlin focuses on navigational queries and provides many constructs to specify and traverse paths. Traversals are at the core of every query and we specify path-patterns via steps on these traversals. Apart from simple, non-repeating path-patterns, Gremlin also supports arbitrary or fixed iterations of traversals via the `repeat()` step. Therefore, Gremlin supports full RPQs and is even more expressive as it allows iterations of whole traversals [AAB+18]. We can also return paths in their own data structure, which is essentially a list of objects and therefore similar to Cypher. However, as the data model does not support paths, we cannot augment them with properties or store them. [AAB+17]

Reachability query 7 asks for the length of the shortest path between two `person` nodes. Gremlin does not provide a function that selects such a path but we can find notes on this on the Recipes page on the TinkerPop website[18]. Before we limit ourselves to the shortest path in Algorithm 4.36, we start with a description of the path in general. Starting at the node representing one person, we use `repeat()` together with `until()` to specify a path of arbitrary length over `knows` edges. The `simplePath()` step ensures that the traversers do not repeat their paths, therefore avoiding cycles. Gremlin also provides a `path()` step that does not restrict the traversers and a `cyclicPath()` step that ensures that the traversers do repeat their paths. In our case, we loop until we find the vertex with the given `iid`. We use `path()` to get that path and count the number of vertices on it via `count()`. Since our understanding of a path length usually regards the number of edges, we subtract 1. This query selects a single shortest path as we pick the first one via `next()` and the first path will always be a shortest one in Gremlin.

Note however that this implementation cannot deal with the two special cases given in the definition of the query: if `person1Id=person2Id`, this implementation will not return 0 but search for a path with length at least 1. However, this case can be easily handled

[18] `https://tinkerpop.apache.org/docs/current/recipes/#shortest-path`

in the programming language by a comparison of the two parameters. Furthermore, if the two `persons` are not connected over `knows` edges, the query will either test all simple paths or run into a timeout depending on the concrete implementation of the language-variant. Whereas some versions of Gremlin provide access to the number of iterations via `loops()` and therefore allow for a bounded repetition, we did not manage to use this function in our version.

```
1 length = g.V().has("person", "iid", person1Id).
2   repeat(out("knows").simplePath()).until(has("person", "iid", person2Id)).
3   path().count(Scope.local).next();
4 length = length - 1;  //as length contains the number of vertices
```

Listing 4.36: Parts of Query 7 (IC13) as a Gremlin query.

The query in Algorithm 4.37 integrates the pattern matching functionality of Gremlin with a path-query. We start with a selection of the person and move up to three times (the argument of `times()` specifies an upper bound) over an outgoing `knows` edge. In line 3, we remove duplicates via `dedup()` to avoid cycles and limit ourselves to nodes with the given `firstName`. Now that we have a reference `p` to those persons, we switch to the declarative pattern matching style and gather more information related to `p`.

If we compare this query to its implementation in Cypher in Algorithm 4.17, this version looks quite compact. However, this query does not fully match the definition of the query given in the benchmark and is only given to get a feeling for such scenarios[19]. The implementation given here for example does not include the distance between the starting person and the person with the given `firstName`. Furthermore, the query below matches only on those nodes `p` where the complete pattern matches, i.e. when a node `p` has outgoing `isLocatedIn`, `workAt` and `studyAt` relationships. As not every person has to have work and study related information in our data model, we used `OPTIONAL MATCH` for this in the Cypher query. However, there is no step in Gremlin that matches a pattern only if it exists. We can achieve this functionality by splitting the query into multiple ones, where the first one collects the friends with the given first name and their location. Passing this information on to subsequent queries, we can then check whether work and study related information exists and, if applicable, collect this.

```
1 g.V().has("person", "iid", personId).
2   repeat(out("knows")).times(3).
3   dedup().has("firstName", firstName).as("p").
4   match(
5     __.as("p").out("isLocatedIn").as("locationCity"),
6     __.as("p").out("workAt").as("company").out("isLocatedIn").as("cc"),
7     __.as("p").out("studyAt").as("uni").out("isLocatedIn").as("uc")
8   ).select("p", "locationCity", "company", "cc", "uni", "uc").by(valueMap());
```

Listing 4.37: Parts of Query 8 (IC1) as a Gremlin query.

[19] As for the other queries, an implementation of this query as it matches the benchmark's definition can be found in our GitHub repository. This implementation however is notably longer with around 150 lines of code, uses multiple, more sophisticated Gremlin queries and aggregates all information in Java.

The query in Algorithm 4.38 does not contain any new concepts apart from the `limit()` step that, as the name suggests, limits the number of traversers from that point on. We use this step to output information regarding the 10 most recent `messages` from the user, by first sorting them by their `creationDate` in descending order and then limiting the result to 10 entries. Again, the query as given here cannot deal with a special case, namely that one of the most recent `messages` is a `post` and not a `comment`. In that scenario, the query would not match the `message` at all as a `post` has no outgoing `replyOf` edge. We could achieve this by splitting it into multiple queries where we first select the 10 most recent `comments` or `posts`, and gather the rest in another query depending on their type.

```
1 g.V().has("iid", personId).
2   in("hasCreator").as("message").order().by("creationDate", decr).
3   repeat(out("replyOf").simplePath()).until(hasLabel("post")).as("post").
4   out("hasCreator").as("op").
5   select("message", "post", "op").by(valueMap()).by("iid").by(valueMap("iid",
      "firstName", "lastName")).
6   limit(10);
```

Listing 4.38: Partial implementation of Query 9 (IS2) in Gremlin. Note that this implementation cannot deal with the case that a recent `message` is a `post`.

Evaluation Semantics

Under default settings, Gremlin uses arbitrary path semantics where paths in the evaluation are not restricted. Therefore, the evaluation of a Gremlin query can contain an infinite number of paths but also paths of infinite lengths. As briefly mentioned before, we can also switch to no-repeated-node semantics using the `simplePath()` step that avoids non-terminating traversals. We have also mentioned the `cyclicPath()` step that uses a special semantics where only traversers that repeat any of their entities (nodes or edges) on their path are kept. That does not avoid non-terminating traversals as such a query can forever loop over a cycle in the graph if the length is not restricted. Regarding possible outputs of queries, Gremlin behaves very similar to Cypher as it supports fixed arity outputs like single values, properties or entities as well as variable arity outputs like paths and lists. [DXWL19]

4.2.4 Data Manipulation

A node can be inserted in a traversal using the `addV()` step that takes a list (or array in Gremlin-Java) as input and creates a node. The array can be used to set the label, id and properties of the entity at creation time as we can see in Algorithm 4.39. Properties can be added to an existing entity either via a reference as in line 4 or in a traversal as in line 5. Similarly, we can remove a property either from a reference (line 6) or in a traversal (line 7). Note however that there are two different steps for this: whereas properties and entities are deleted via `remove()` when using a reference, `drop()` provides the same functionality for traversals. Regarding the insertion of edges (lines 9-10), we first select the target node and connect the existing node to the target via the `addEdge(<label>,`

<target>) step. Note that this can also be done in a traversal but since the syntax for this differs heavily between the version we are using and newer ones, we omit such an example and refer the reader to the documentation. The query given below depicts only a small part of Query 10 and the full query is again given in our GitHub repository.

```
1  Object[] keyValues = new Object[]{T.label, "person", "iid", personId, "
       firstName", firstName};
2  Vertex person = g.addV(keyValues).next(); //store reference to vertex
3
4  person.property("lastName", lastName);     //add property via reference
5  g.V(person.id()).property("cd", "test");   //add property via traversal source
6  person.property("lastName").remove();      //remove property via reference
7  g.V(person.id()).properties("cd").drop(); //remove property via trav. source
8
9  Vertex place = g.V().has("iid", cityId).next(); //get target vertex
10 person.addEdge("isLocatedIn", place);      //add edge
11 [...]
```

Listing 4.39: Simplified version of Query 10 (INS1) in Gremlin that inserts only some properties and a single edge.

Unlike Cypher, there is no way in Gremlin to remove a vertex without also removing its incident edges. To that end, the `drop()` step, when called on a vertex, automatically removes all incident edges together with the vertex. This step is also used to remove properties as we have seen above, and can also be used to remove edges. As we can see in Algorithm 4.40, the syntax is notably more complicated compared to Cypher's when triggering the removal of adjacent entities. In this example, we select the initial comment in the first line and pass this through `union()` via the identity function `__()` in line 3. We repeatedly move to adjacent vertices that are connected over incoming `replyOf` edges and emit them such that all those vertices are contained in the result of `union()`.

```
1 g.V().has("iid", commentId).
2   union(
3     __(),   //identity function to pass the node with iid=commentId
4     repeat(in("replyOf")).emit()  //emit all replies, not only the leaves
5   ).drop();
```

Listing 4.40: Query 11 (DEL7) as a Gremlin query.

Properties can be updated the same way as they are inserted: using the `property()` step. When evaluating `property("key","value")`, the system will first remove all properties under the `key` on the given entity before the new key-value property is added. Therefore, the query in Algorithm 4.41 can simply set the new property without having to delete the previous value.

```
1 g.V().has("iid", personId).outE("studyAt").as("e").
2   otherV().has("iid", univId).
3   select("e").property("classYear", "2011")
```

Listing 4.41: Query 12 as a Gremlin query.

On a side node, Gremlin supports multi-value properties on vertices, meaning that a vertex can have multiple property values under the same `key`. When adding or updating such a property using the version of Gremlin in Titan, we can specify that the values under a given key are stored as a list (allowing duplicates), as a set (no duplicates) or as a single value (ruling out multi-values on that property key). Apart from multi-value ones, Gremlin also supports meta-properties, that are properties on properties. This can for example be used to store access permissions on certain properties that can be checked by an application as we can see in Algorithm 4.42.

```
1 v = g.V().has("name", "John Doe").next();
2 g.V(v).properties("name").property("permission", "all"); //set permission
3 g.V(v).properties("name").hasValue("permission", "all"); //check permission
```

Listing 4.42: Example of the creation and access to meta-properties in Gremlin.

4.2.5 Data Definition

The data model used by Gremlin is a directed, attributed multi-graph where each entity can only have a single, immutable label. Therefore, we cannot change the label of an existing edge or vertex after its creation. Gremlin also allows us to connect to multiple graphs. A traversal however can only access a single graph as it is spawned from a traversal source (`g` in our examples), that in turn is associated with a graph as we have seen in Algorithm 4.23. Regarding a schema, Gremlin as specified by the TinkerPop project does not provide any steps or functions to view, create, update or delete a schema and is therefore completely schema-less. Each vendor that integrates the TinkerPop framework to become TinkerPop enabled can choose whether a schema is required, supported, or if it is a schema-less system. To this end, a vendor can provide functions to access and modify a schema or other constraints.

4.2.6 Further Features

- **Composability:**
  The version of Gremlin used in our examples does not support multiple occurrences of either the `V()` or `E()` steps and therefore no (linear) composition of queries. However, there is support for this in newer versions as in lines 1-2 in Algorithm 4.43. We can also nest multiple traversals as we can see in lines 4-6, where `next()` triggers the evaluation of the inner traversal (line 5) that returns a list of strings which in turn is used as parameter in the outer one that returns all persons with the same `firstName` as the person with id `102` or `103`.

```
1 g.V().has("name", within("John", "Mary")).as("person").
2   V().hasLabel("software").as("software").select("person","software")
3
4 g.V().has("firstName", within(
5     g.V().has("iid", within("102","103")).values("firstName").fold().
    next())
```

```
6 )
```

Listing 4.43: Example of a query using multiple `V()` steps (as it is supported in newer version of Gremlin) and of a query with a nested traversal.

As briefly mentioned before, we can also nest many steps, as a step is essentially a function and Gremlin supports, and even relies on, function composition and nesting. Apart from such applications where we convert the output for example to a string or a list of strings and compare properties, Gremlin queries are generally not composable. That restriction might come from the design of the language, where a query is represented by a traversal that is lazily evaluated by default. Furthermore, the setting of Gremlin as a language that is implemented as language variant and embedded into a programming language places the query language in a different position compared to most other query languages. That for example allows us to split a bigger query into smaller ones, save the outputs in data structures of the programming language and even outsource some computation or query logic to the programming language. This allows for a completely different way of writing queries, composing them and relaying their logic to one another.

- **Paths as first-class citizens:**
  Similar to Cypher, Gremlin provides the possibility to output paths in their own data structure. However, paths are not raised to first-class citizens as there is no support for them in the data model which means that we cannot save them in the model.

We have seen some of the OLTP functionality of Gremlin and its rather unusual approach as a low-level query language that is more imperative and merges with the implementing programming language. This approach however allows vendors to augment the language with further functions that are not part of the version in the TinkerPop framework. If we look at Gremlin as a domain specific language built to traverse graphs, and using the fact that we can implement our own language-variant, a natural extensions leads to *Domain Specific Languages (DSL)* that abstract from the underlying graph structure to the domain itself. As an example, we can write a social DSL that abstracts from the general structure of a graph to the domain of a social network. This allows us to introduce domain specific steps that allow for more compact queries as we can see in Algorithm 4.44. Here, we assume that the DSL provides two new steps: `persons()` that starts traversers on the nodes, filters on their label and on the given `name`, and `knows()` that traverses to adjacent `persons` and filters on the given `name`.

```
1 g.V().has("person", "name", "John Doe").out("knows").has("person", "name",
    "Mary Ann")
2 g.persons("John Doe").knows("Mary Ann")
```

Listing 4.44: Example of two equivalent queries: first in Gremlin and then in a hypothetical social DSL.

Apart from this, Gremlin provides other interesting functions that set it apart from many graph query languages. The `subgraph()` step, that produces an edge-induced subgraph from a traversal, is such an example. One could think that this step makes Gremlin at least somewhat composable, but this is not the case as a traversal source can only query a single graph. In other words, we have to create a new traversal source on the subgraph before it can be queried. Most functions of the language that we mentioned are especially relevant in OLTP workloads. However, the imperative nature of Gremlin makes it "*more suitable for expressing graph analysis algorithms such as PageRank, Betweenness Centrality, etc*" [vRHK+16]. To this end, there is a dedicated API in the TinkerPop framework that provides many such functions. As we focus on OLTP workloads, we do not look into these functions in more detail and refer the reader to the documentation for more information on this.

4.3 PGQL

We have already seen a declarative query language that aligns closely with SQL: Cypher. The Property Graph Query Language (PGQL) was inspired by both, SQL and Cypher, and takes the alignment to SQL a step further as it "*aims to follow SQL syntax where possible*" [Pla18]. This does not come as a surprise as the language is developed by Oracle and offered in some of their products. Unlike Cypher queries that are conceptually structured linearly, PGQL queries use the select-from-where structure inherent to SQL. As PGQL is not standardized, Oracle aims to keep the language in sync with the standardized features of SQL by providing the same query structure, the same clauses and most functions [HLP+19]. As a side note, there is also ongoing work inside ISO/IEC JTC 1/SC 32's working group 3, the same group that is working on the new GQL standard, to add graph querying functionality to SQL itself under *SQL/PGQ*. Although there is no final specification of this functionality as of now, PGQL aims to provide pattern matching capabilities that roughly resemble the ones on this extension [HLP+19]. PGQL, among other languages like Cypher or G-CORE influence the creation of this addition and it is therefore possible that a future version of SQL has capabilities similar to PGQL regarding graph queries.

Similar to SQL, a query in PGQL is at least composed of a `SELECT` and `FROM` clause that can be followed by optional clauses like `WHERE`, `ORDER BY` or `GROUP BY` [SHvR+16]. Graph patterns form the basis of PGQL and can be specified in the `FROM` clause using one or more `MATCH` clauses [vRHK+16]. As patterns are specified using an ASCII art representation similar to Cypher, these `MATCH` clauses closely resemble their counterparts in Cypher [FGG+18b]. The `SELECT` clause in turn resembles its counterpart in SQL as it allows users to specify the projections or aggregations that are contained in the output of a query. Algorithm 4.45 shows a simple PGQL query and its select-from-where structure that matches a single node `p:Person`.

```
1 SELECT p.firstName, p.lastName
2   FROM MATCH (p:Person)
3   WHERE p.id = 123
```

Listing 4.45: A simple PGQL query that retrieves the first and last name of the person with a given id.

Another similarity between PGQL and SQL as well as Cypher regards the evaluation of a query. We have already mentioned that a Cypher query outputs a table containing bindings form query variables to elements. Oracle's PGQL works analogously as it returns tables containing variables and their bindings. These tabular result sets closely resemble the ones from SQL queries, which allows PGQL queries to be nested inside SQL queries [vRHK+16]. As the graph query language extends the relational query language also with graph specific data types like `vertex, edge` and `path`, variables of these types can be included in the output. Nonetheless, the output is always structured in a tabular form and such graph specific types can only be included as entries in that table. Using these graph related types, we can for example output vertices and their ids and labels via the query in Algorithm 4.46. On a side note, vertices as well as edges can be augmented with none, one or multiple labels in the data model used by PGQL. Assuming that an entity has only a single label, we can use the `LABEL` function to retrieve it. However, this function produces an error on entities with none or multiple labels. The `LABELS` function on the other hand produces a set of labels and can deal with all cases by returning an empty set on entities without labels and a set containing the label(s) on entities with at least one.

```
1 SELECT id(p), labels(p) AS lbl, p FROM MATCH (p:Person) LIMIT 5
```

Listing 4.46: A PGQL query that selects the ids, labels and the vertices themselves of 5 `persons`.

When looking at the `MATCH` clauses in the queries in Algorithm 4.47, we can see the close resemblance to the syntax of patterns in Cypher. Similar to Cypher, anonymous nodes are depicted via empty `()` brackets and anonymous edges via empty `[]` brackets or by omitting them altogether. Furthermore, edges in the data model used by PGQL have to be directed but we can match edges in both directions using an any-directed edge pattern ("`-[]-`" or "`- -`") as in the fourth query. In contrast to Cypher however, there is no compact syntax to specify restrictions on entities like in Algorithm 4.2, as this would not be compatible with the goal of following the SQL syntax where possible.

Apart from the pattern matching part in the `MATCH` clause(s), most of the functions and clauses used outside of `MATCH` are coming directly from SQL. That includes for example the dedicated `GROUP BY` clause as we can see in the first two queries [20]. Another resemblance to SQL regards the composition of queries using either existential subqueries via `(NOT) EXISTS` or scalar subqueries as in the third and fourth query. If we compare the fourth query to the implementation of the same query in Cypher in Algorithm 4.4,

[20] Remember: Cypher for example takes a different approach to grouping as the result is either grouped implicitly or on the key specified in the `WITH` clause.

we can observe the difference between the conceptually linear structure in Cypher where queries can be evaluated linearly from top to bottom, and the structure of PGQL and SQL where inner queries are often evaluated before outer ones. Furthermore, we can parameterize queries using a bind variable "?" as we do with the `lastName` in line 16.

```
1  SELECT p.age, COUNT(*) FROM MATCH (p:Person) GROUP BY p.age
2
3  SELECT label(e) AS lbl, COUNT(*)
4    FROM MATCH () -[e]-> ()      /*anonymous nodes*/
5    GROUP BY lbl ORDER BY COUNT(*) DESC
6
7  SELECT p.name AS name,
8      (SELECT COUNT(o) FROM MATCH (p) -> (o)) AS outgoingConnections,
9      (SELECT COUNT(DISTINCT(o)) FROM MATCH (p) -[:KNOWS]- (o)) AS knows
10   FROM MATCH (p:Person)
11
12 SELECT o.firstName, o.lastName
13   FROM MATCH (m) -- (o:Person) /*short form of anonymous, any-directed edge*/
14   WHERE m.firstName IN (
15     SELECT m1.firstName FROM MATCH (p:Person) -- (m1:Person)
16     WHERE p.firstName = 'John' AND p.lastName = ?  /*bind variable*/
17     ORDER BY p.firstName DESC LIMIT 1)
```

Listing 4.47: Four PGQL queries that show the resemblance between PGQL and SQL.

As PGQL is only implemented in some research projects and commercial products of Oracle, we did not manage to test the queries in this chapter on a database. However, we used a parsing software from Oracle[21] to ensure that the queries are syntactically correct.

4.3.1 Structure Independent

We omit the code of Queries 1 and 2 here as both of them use only a very simple pattern matching part that matches a single node and the rest of the query is SQL code. To implement Query 2, we use the `CASE` construct that we will also see in Algorithms 4.49 and 4.54. Regarding Query 3, we run into a problem of the data model used by PGQL as, similar to Gremlin's data model, it does not support lists. Therefore, we cannot store the languages spoken by a person in a list in the first place. We could concatenate the languages in a string, store this string and split it up again when retrieving it to count the number of entries for this query. As this would be done in a programming language and not in PGQL, we cannot implement this query directly in the query language.

If we look at the functions provided by PGQL to aggregate data and work with numeric or other data types, we find many functions that are also available in SQL. We can for example use `SUM`, `COUNT`, `AVG`, `ABS`, `CEIL(ING)`, `FLOOR`, `ROUND` on numeric data types but also a regex checker on strings. PGQL furthermore adds vertex and edge functions like `LABEL(S)`, `ID`, `IN_DEGREE` and `OUT_DEGREE` where the latter ones are not structure independent but nonetheless interesting in a graph query language.

[21] https://github.com/oracle/pgql-lang

4.3.2 Pattern Matching

We have already seen parts of the pattern matching functionality and the resemblance to the Cypher syntax in the previous examples. As Query 4 contains only a very basic graph pattern, we omit its implementation and continue with the query in Algorithm 4.48. The pattern in this query connects a `Person` to some node over a single `knows` edge. We can see in line 1 that edges can have properties and we can access them in the same way as properties on vertices. Furthermore, we can order the result using the same syntax as in SQL: in an `ORDER BY` clause.

```
1 SELECT f.id AS personId, f.firstName, f.lastName, e.creationDate AS
     friendshipCreationDate
2   FROM MATCH (p:Person) -[e:knows]-> (f) WHERE p.id = ?
3   ORDER BY friendshipCreationDate DESC, personId ASC
```

Listing 4.48: Query 5 (IS3) as a PGQL query.

To implement Query 6, we have to check for the existence of an edge between two nodes. While we use the `OPTIONAL` clause for this in our Cypher implementation, there is no equivalent clause in Gremlin and we could not implement this in a single query. PGQL also lacks a dedicated clause for this but we can simulate such an optional pattern using an optional path expression [AAB+18]. As we can see in Algorithm 4.49, we specify the fixed pattern in the first `MATCH` clause in line 3 and use a second `MATCH` for the optional pattern. In our case, the connection is optional which is depicted by the question mark after the edge. We test for the existence of this edge using a `CASE` predicate in line 2 and, depending on that result, set a value for the `knowsAuthor` variable. This implementation shows that there can be multiple correlated `MATCH` clauses in a single query. PGQL also supports uncorrelated ones that are evaluated as a Cartesian product of the individual evaluations.

```
1 SELECT c.id, c.content, c.creationDate, p2.id, p2.firstName, p2.lastName,
2   CASE k IS NOT NULL WHEN true THEN true ELSE false END AS knowsAuthor
3   FROM MATCH (p1:Person) <-[:hasCreator]- (m:Message) <-[r:reply_of]- (c:
      Comment) -[:hasCreator]-> (p2:Person),
4         MATCH (p1) -[k:knows]-? (p2)
5   WHERE m.id = ?
```

Listing 4.49: Query 6 (IS7) as a PGQL query.

Evaluation Semantics

Earlier versions of PGQL used a no-repeated-node semantics as default and included special keywords to switch to a homomorphism-based semantics [vRHK+16]. From version 1.0 onward, a homomorphism-based semantics is used as default as it translates better to and from SQL, among other things [vR17]. Nonetheless, we can switch to an isomorphism-based semantics using the `ALL_DIFFERENT` predicate. As this predicate can deal with vertices as well as edges, we can achieve no-repeated-node, no-repeated-edge and even no-repeated-anything semantics.

4.3.3 Path

The graph query language extends SQL not only with a pattern matching functionality but was also the first property graph query language with full support of RPQs [Pla18]. To this end, users can specify named path pattern macros at the beginning of a query. These patterns can be referenced later on in the query, either when specifying other path pattern macros or in a `MATCH` clause. Such named path patterns were first proposed in PGQL [FGG+18b] and can contain arbitrary regular path patterns and also an optional `WHERE` clause. The `WHERE` clause allows us to not only access labels on the path but also specify requirements on properties of both, vertices and edges [vRHK+16]. This is needed for more sophisticated patterns in the property graph data model as we otherwise cannot compare properties along a path. On the other hand, this exceeds the functionality of RPQs and we land in the area of REMs that allow for exactly that, namely to compare data values on a path. Nonetheless, PGQL supports only a subset of REMs as it is "*hard to come up with a syntax for REMs that is declarative*" [vR17] and PGQL is a declarative language. Since PGQL furthermore provides a `path` data type that allows us to outputs paths, the query language also supports ECRPQs [Bar13].

Algorithm 4.50 contains a query that uses a path pattern macro. The pattern is specified and named `eq_voltage_hop` in the first line and used in the third one via a reference on that name. Forward slashes "`-//-`" instead of square brackets "`-[]-`" on edges denote the reachability semantics. This semantics is useful when we are only interested in whether there exists at least one path between two vertices and not in all those paths. We will see later on that we can also use this semantics on other, non-macro, paths. However, path pattern macros can only be used with this semantics. In this query, we use a functionality of REMs as we compare the `voltage` properties of nodes on the paths.

```
1 PATH eq_voltage_hop AS (n:Device) -> (m:Device) WHERE n.voltage = m.voltage
2 SELECT y.name
3   FROM MATCH (x) -/:eq_voltage_hop+/-> (y)
4   WHERE x.name = 'generator_x29'
```

Listing 4.50: A PGQL query using a path pattern macro with data comparison. The query selects all devices reachable from `generator_x29` via a path where each device has the same `voltage`. Source: *GQL - Status Update And Comparison to LDBC's Graph QL proposals*, slide 21 [vR17].

Although this is a functionality that sets PGQL apart from many other graph query languages, we will not use it in the following queries as we do not need them for the SNB queries. As we can see in Algorithm 4.51, PGQL provides a `SHORTEST` function that matches a single shortest path that satisfies the pattern. We can compute the length of the path by counting the number of edges on it as we do in line 1. In this query, we are not sure if such a path exists at all, which is why we use a `CASE` expression to test whether the variable `e` is `null`[22].

[22]Note that the specification does not state if `e` would indeed be `null` if there is no such path. This would have to be tested and we can only assure that the query is syntactically correct.

```
SELECT CASE e IS NULL WHEN true THEN -1 ELSE COUNT(e) END AS
    shortestPathLength
  FROM MATCH SHORTEST ( (p1:Person) -[e:knows]-* (p2:Person) )
  WHERE p1.id = ? AND p2.id = ?
```

Listing 4.51: Query 7 (IC13) as a PGQL query.

Apart from querying for a single shortest path, we can also query for the "`TOP K SHORTEST`" ones as in Algorithm 4.52. This query furthermore specifies a restriction on the `weight` value(s) of the edges on the path directly inside the pattern. As a side note, PGQL also provides cheapest path finding based on a cost function via "`(TOP K) CHEAPEST`".

```
SELECT src, ARRAY_AGG(e.weight), dst
  FROM MATCH TOP 3 SHORTEST ( (src) (-[e]-> WHERE e.weight > 10)* (dst) )
```

Listing 4.52: A PGQL query that queries for the 3 shortest paths between `src` and `dst` where each edge along the path has a `weight` greater 10. Source: *PGQL 1.3 Specification* [Ora20].

As mentioned when we analyzed Query 3, PGQL does not support lists. There is however one exception for query outputs: the horizontal aggregation function `ARRAY_AGG` aggregates the given values into an array/list. Whereas vertical aggregation functions work similar to ones in SQL, where a group of values from different rows are aggregated, horizontal functions aggregate over a path. However, we cannot aggregate over arbitrary variables or properties on a path which would be needed to collect study- and work-related information in Query 8. Algorithm 4.53 contains some parts of the query that we will briefly go through. We use `ARRAY_AGG` to show how we can aggregate multiple occurrences of the same property, for example university names, in a list. Furthermore, `{1,3}` specifies a lower and upper bound for the number of `knows` edges that can be traversed. Apart from such specific bounds, we can use "`*`" for paths of any length (including 0), "+" for paths of length at least 1 and "`?`" for paths of length 0 or 1.

```
SELECT /* [...] */
  ARRAY_AGG(u.name) AS friendUNames, ARRAY_AGG(es.classYear) AS friendUYears,
     ARRAY_AGG(uc.name) AS friendUCities
  FROM MATCH SHORTEST ( (p:Person) -[e:knows]-{1,3} (f:Person) ),
  MATCH (f) -/es:studyAt?/-> (u:University) -/:isLocatedIn/-> (uc:City),
/* [...] */
```

Listing 4.53: Parts of Query 8 (IC1) as a PGQL query.

Algorithm 4.54 contains a simplified version of Query 9 where we integrate reachability semantics into a larger pattern. Furthermore, this algorithm and also the previous one show that we can use reachability not only on a path pattern macro but also on simpler paths directly given in `MATCH`.

```
SELECT m.creationDate AS creationDate, po.id, op.id, op.firstName,
```

```
2     CASE m.content IS NOT NULL WHEN true THEN m.content ELSE m.imageFile END,
3   FROM MATCH (p:Person) <-[e:hasCreator]- (m:Message) -/:replyOf*/-> (po:Post
      ) -[:hasCreator]-> (op:Person)
4   WHERE p.id = ?
5   ORDER BY creationDate DESC LIMIT 10
```

Listing 4.54: Simplified version of Query 9 (IS2) as a PGQL query that selects only some properties.

Evaluation Semantics
PGQL avoids paths of unlimited length and an unlimited number of paths in the evaluation by allowing pattern quantifiers only when using reachability semantics or inside `SHORTEST` patterns. Therefore, quantifiers like "`*`" or "`+`" but also quantifiers with custom lower and upper bounds can only be used inside reachability demarcated paths, where they are given inside the demarcation: "`-/var:label<quantifier>/-`", or using standard path syntax inside "`(TOP K) SHORTEST`", where they are given outside the demarcation: "`-[var:label]-<quantifier>`". We have already mentioned that PGQL also provides cheapest path finding but this also limits the found paths either to a single one or to the top k. Apart from these built-in functions that use either a shortest or cheapest path semantics, we can implement our own semantics and use it in PGQL via a user-defined function.

4.3.4 Data Manipulation

With the introduction of version 1.3 in March 2020, PGQL introduced the graph modification clauses `INSERT`, `UPDATE` and `DELETE`. In earlier versions, the query language was limited to DQL queries. Algorithm 4.55 depicts a simplified version of Query 10 where we omit most properties for brevity. Compared to the Cypher implementation of the query in Algorithm 4.18 where we could pass a list of Ids and unroll them inside the query using `UNWIND`, there is no such functionality in PGQL. Therefore, we cannot pass a variable number of tag IDs for example. To achieve this functionality, we have to write multiple queries or include a single `EDGE` clause for each edge that is to be inserted. Either way, the query has to be adapted depending on the number of edges that are inserted, which is not necessary in Cypher where we can use the same query to insert a variable number of edges. Apart from that, the query uses the `DATE` and `TIMESTAMP` types that are available next to `TIME` and further date/time related types for time zones.

```
1 INSERT VERTEX p LABELS (Person)
2     PROPERTIES (p.id=1234, p.birthday=DATE '1990-01-01', p.creationDate=
      TIMESTAMP '2020-01-01 16:15:00'),
3   EDGE e BETWEEN p AND c LABELS (isLocatedIn),
4   EDGE ew BETWEEN p AND co LABELS (workAt) PROPERTIES (ew.workFrom=1990),
5   EDGE et BETWEEN p AND t LABELS (hasInterest)
6   FROM MATCH (c:City), MATCH (co:Company), MATCH (t:Tag)
7   WHERE c.id=? AND co.id=? AND t.id=?
```

Listing 4.55: Simplified version of Query 10 (INS1) in PGQL that inserts only some properties.

Similar to Gremlin, the removal of a vertex also triggers the removal of all incident edges of that vertex. Therefore, we can remove a comment, all incident edges and replying comments via the query in Algorithm 4.56.

```
1 DELETE c, r
2   FROM MATCH (c:Comment) <-/:replyOf*/- (r:Comment)
3   WHERE c.id = ?
```

Listing 4.56: Query 11 (DEL7) as a PGQL query.

Properties can be updated similar to SQL using the `UPDATE` clause in combination with `SET`. As the implementation of Query 12 combines functions of SQL and Cypher, we omit it here and refer the interested reader either to our GitHub repository or the PGQL specification.

4.3.5 Data Definition

PGQL uses a property graph data model with support for multiple graphs. Each graph has a name and we can specify which graph we want to query in the `ON` clause. This clause can be omitted if a default graph is set, which however cannot yet be done through PGQL itself but only through APIs on the database system. Each `MATCH` clause can be augmented with an optional `ON` that specifies the graph that the pattern is matched on as we can see in Algorithm 4.57. In all our queries, we assume that we are working on a single graph that is set as default graph. Each property graph consists of vertices and edges that can have zero or more labels and zero or more properties. As mentioned before, edges in that data model have to be directed.

```
1 SELECT p.name, q.name, a.number
2   FROM MATCH (p) -[:knows]-> (q) ON social_graph
3        MATCH (a:Forum) ON other_graph WHERE a.name='test'
```

Listing 4.57: A PGQL query that queries for two persons from the `social_graph` and a forum from the `other_graph`.

Similar to Cypher and Gremlin, PGQL does not support a schema. However, we can create a graph out of data from existing tables via the "`CREATE PROPERTY GRAPH`" statement. Using this statement, we specify the source tables for vertices and edges, the labels for the entities and their properties as well as the way they will be connected via edges. In other words, if we assume that there are tables representing vertices and tables representing edges, we can use this statement to build a property graph from that data. Therefore, we specify the graph schema that is created from the tabular data, but this schema is only used once to create the graph, and not enforced later on when interacting with it using DML queries. This setting, where data from (relational) tables is used to create a property graph, makes sense if we look at the position of PGQL in Oracle's products. There, it is used to query property graphs where the underlying data is often[23]

[23] Using PGX, the system can switch to a native graph storage when the data is stored in-memory. In this setting, it effectively becomes a native graph storage.

stored in a relational database [BPG+19]. In other words, PGQL provides a graph-view of data that is stored in relational tables. Algorithm 4.58 contains an example of such a statement that creates a property graph named `social_network` with data from 5 tables. The graph will be structured as specified in Query 13, where a `person` is `livingat` a `street` which in turn `islocatedin` a `city`.

```
1  CREATE PROPERTY GRAPH social_network
2    VERTEX TABLES (
3      persons
4        LABEL person PROPERTIES (person_id, firstname, lastname),
5      streets LABEL street PROPERTIES (street_id, name, length),
6      cities LABEL city PROPERTIES ARE ALL COLUMNS
7    )
8    EDGE TABLES (
9      livingat     /* table name */
10       SOURCE KEY (person_id) REFERENCES persons
11       DESTINATION streets LABEL isLocatedIn PROPERTIES (housenumber),
12     islocatedin  /* table name */
13       SOURCE KEY (streets_id) REFERENCES streets
14       DESTINATION cities NO PROPERTIES
15   )
```

Listing 4.58: Parts of Query 13 as a PGQL query. Note that this creates a new property graph as there is no way to adapt an existing graph.

4.3.6 Further Features

- **Composability:**
  As we have seen in some of our examples, PGQL supports a composition of queries through subqueries similar to SQL. We have also mentioned that a query returns a table containing values. Although these tables can contain graph specific types like vertices, edges and paths, these values are always contained in the table. As a PGQL query takes a graph as input and produces a table, it follows that the query language is not composable [AAB+18].

  In the setting of PGQL as a query language that is used almost solely in the environment of Oracle's databases, which are mostly relational ones, it is unlikely that a future version of the language discards the tabular result format as this would entail that PGQL queries cannot be nested inside SQL queries any more. On the other hand, the paper that introduced PGQL mentions a `graph` data type that "*allows for graph construction and query composition*" [vRHK+16]. Graph construction denotes constructs that allow us to create a graph from one or multiple existing ones. According to the paper, we can construct a graph in the `SELECT` clause. This in turn would mean that such a query returns a graph, and indeed, the paper mentions that a (sub)query can return a `graph`. Unfortunately, there is no trace of such constructs or the `graph` type in the language specifications and the parser did not accept such constructs.

- **Paths as first-class citizens:**
  By providing path pattern macros, where users can specify and name complex path patterns, and allowing for data comparisons along paths PGQL is more expressive than Cypher regarding the path querying capabilities. Although we can name paths, compare them to some extent and output them, they are not raised to first-class citizens as we cannot store paths on their own.

As we have mentioned in the introduction and seen in the examples, we can describe PGQL as a query language that follows the SQL syntax on non-graph related parts and Cypher syntax on most graph related parts of a query. With the addition of "Oracle Spatial and Graph" (that includes PGQL as graph query language) in version 12 of the Oracle Database, many SQL users can use these features in their Oracle databases. On the other hand, the language lacks standardization and support by other vendors [TPAV19]. That can also be seen when searching for information about the query language online, where we are mostly limited to Oracle's own website and we have found no community around this language on any forum or mailing list.

4.4 GSQL

GSQL is the query language used in the TigerGraph database. Similar to PGQL, it is heavily inspired by SQL and extends the relational query language to graph databases by allowing graph-specific primitives in the `FROM` clause [RH19]. Queries that do not use such graph-specific primitives become standard SQL queries [DXWL19]. To achieve this, the syntax of SQL is extended mainly in the `FROM` clause where users can specify graph patterns and navigational path patterns [DWX18]. Furthermore, GSQL uses the type system from standard SQL and extends it with `vertex`, `edge` and `graph` types. These types are needed as we have to declare the schema using a labeled property graph data model before we can either import data or start querying. Such a schema defines the vertex and edge types, their properties and the types of the properties and how entities are related to each other [Tig]. The schema is stored apart from the data graph and is enforced, meaning that we cannot update or insert data that would violate the schema. Therefore, the schema of a graph in GSQL has the same closed world property as a schema in SQL.

GSQL relies on such a strict schema definition as it is especially designed to support large graphs containing billions or trillions of nodes and edges. This comes from the design principle of the TigerGraph database as a "*native parallel graph database*" [DXWL19] for "*tomorrow's big data and analytics*" [HLP+19], where performance on large graphs is important. To support these potentially huge graphs, the database allows a graph to be distributed over multiple machines. Queries on distributed graphs are processed in parallel on the machines before the local values are aggregated. This approach resembles the MapReduce programming model, and GSQL is indeed designed in a way to not only ease the transition for SQL developers but also for NoSQL developers with a MapReduce

mentality [DXWL19]. Coming back to the schema: having this specified beforehand allows the database to use an optimal storage format which in turn results in less space needed to store the data compared to other schema-less databases like Neo4j [RH19]. Furthermore, the schema allows for faster access and is used to plan and optimize query executions in order to achieve high performance parallelism [HLP+19].

As mentioned, the database is especially designed for big data and analytics, and therefore for OLAP workloads. To this end, the query language provides not only OLTP functionality as does SQL, but also iterative and multi-pass algorithms for analytical workloads like PageRank or recommender systems. These algorithms can be implemented using imperative control flow primitives like `IF/ELSE`, `FOREACH` and `WHILE` loops, and queries behave similar to functions as they can invoke other queries or recursively themselves. Together with the support of user-defined functions, GSQL is a Turing-complete language and therefore more expressive than languages like Cypher that are not Turing-complete [WD18, DXWL19]. Despite the focus of TigerGraph on OLAP workloads, we include GSQL in our comparison as it also provides OLTP functionality, transactional graph updates and shows how a query language that uses a graph schema differs from schema-less languages.

General syntax and handling of queries

The query in Algorithm 4.59 shows the select-from-where structure and the graph pattern in the `FROM` clause. Furthermore, we can see that we have to create a query using `CREATE QUERY` before we can execute it. Once a query is created, we can either `INTERPRET` it which results in a line-by-line translation, or `INSTALL` it, where it will be optimized and rolled out to the distributed machines. Furthermore, each installed query has its own REST endpoint that can be used by applications to access the database. In our case, we create a query with the name `ex1` that takes two `STRING` parameters, is specified for the graph with the name `ldbc_snb` and uses the V2 syntax. GSQL's path pattern matching functionality was extended with the introduction of TigerGraph v2.4: where earlier versions under the V1 syntax, that is still the default today, allow for patterns only over a single edge in one `SELECT`, version V2 adds support for multi-hop paths and repetitions. Furthermore, the syntax used to depict a path or an edge differs between the two versions and we will compare them later on.

Apart from the syntax difference, we can see that a GSQL query functions like a container for statements. In this example, the query contains only a single select-from-where (sub)query but as we will see in the next algorithm, a GSQL query can contain multiple SQL `SELECT` statements. Therefore, a GSQL query resembles a stored procedure that can contain multiple `SELECT` statements as well as control-flow primitives like branches and loops [RH19]. When speaking about a SQL (sub)query that is contained in a GSQL query, we will reference the query together with the lines of that query, as in: "the (sub)query in lines 3-5".

Coming back to our example and that subquery in lines 3-5, we note that the pattern syntax in GSQL differs from the ones used by Cypher and PGQL in two major areas. First, parentheses are not used to depict a node, as nodes are in most cases not delimited

at all. Instead, parentheses are used as delimiters on edges. And second, the places of the type and variable name on both, vertices and edges, are swapped. Whereas Cypher and PGQL use a `<variable>:<type>` syntax, GSQL reverses it to `<type>:<variable>` as in `Person:p`.

```
1 CREATE QUERY ex1(STRING fN, STRING lN) FOR GRAPH ldbc_snb syntax v2 {
2
3   friends = SELECT friend
4     FROM Person:p -(KNOWS:k)- Person:friend
5     WHERE p.firstName == fN AND p.lastName == lN;
6
7   PRINT friends [friends.firstName, friends.lastName];
8 }
```

Listing 4.59: A simple GSQL query that selects the friends of a `Person` with a given name.

Accumulators

In order to achieve efficient parallel computation, GSQL introduces the concept of accumulators and the corresponding `ACCUM` clause. Using this clause, we can specify compute functions on nodes and edges that can be executed in parallel in order to aggregate values into accumulator variables. These variables can either be global or attached to a vertex (local). They are typed, hold data and can be written to in parallel. In other words, an accumulator is a mutable mutex variable that is shared among all threads that participate in the execution of a query. The data written to an accumulator is then combined using the specified compute function. GSQL provides many such functions out of the box and users can also provide their own function.

Algorithm 4.60 shows the usage and difference of local and global accumulators as well as simple variables like `localVariable`. `SumAccum<INT>` is a global accumulator (depicted by @@) and therefore visible to all threads. As the name suggests, values written to this variable will be aggregated by summing them. Apart from `SumAccum`, GSQL provides other accumulator types on numbers like `MaxAccum`, `AvgAccum`, logical accumulators like `OrAccum`, `AndAccum` and ones to create collections like `SetAccum` and `MapAccum`. The second accumulator in our example is a local one (a single @), meaning that each vertex selected by the query will have its own `localRelationsCount`. As we can see in line 7, we have to specify the vertex when writing to a local accumulator. In contrast, global accumulators can be directly accessed as in line 8. After the evaluation of lines 6-9, the global accumulator holds the global number of `WORK_AT` relationships and each `Company` vertex holds the local number of incoming `WORK_AT` edges. Note that, intuitively, we do not increase the counts directly by 1 in lines 7 and 8 but we add the number "1" to the accumulator variables and these numbers are then aggregated by summing them. Therefore, we write to all accumulator variables, regardless of them using `sum`, `set`, `map` or a custom function, using the "+=" syntax. Coming back to our query, each process has its own copy of `localVariable` that is not shared. Therefore, each of them will increment the local copy by one and the query will print 1 in line 11.

Apart from the accumulators, the query shows that we can save a reference to entities as we do with the comps variable in line 6. As we select all companies c in this subquery, comps contains a reference to all those vertices. We can use these references to directly access these entities for example to check properties, print values, but also in other FROM clauses as in line 15. Unlike in the query in lines 6-9 where we use a pattern with the Company and Person vertex types and the WORK_AT edge type[24] in the FROM clause, we have no reference to such pre-defined types in the query in lines 14-16. The latter query starts with all entities in the comps variable, orders them by their local accumulator value and saves a reference to the top 10 such vertices again to comps. Therefore, the variable comps contains the 10 companies with the most incoming WORK_AT relationships after that second query and we print those in line 17.

```
1  CREATE QUERY ex2() FOR GRAPH ldbc_snb syntax v2 {
2    INT localVariable;
3    SumAccum<INT> @@globalRelationshipCount;
4    SumAccum<INT> @localRelationshipCount;
5
6    comps = SELECT c FROM Company:c -(<WORK_AT)- Person:t
7      ACCUM c.@localRelationshipCount += 1,    //local, preceded by variable
8            @@globalRelationshipCount += 1,    //global
9            localVariable = localVariable + 1; //local variable
10
11   PRINT localVariable;              //outputs 1
12   PRINT @@globalRelationshipCount;  //outputs the global count
13
14   comps = SELECT c
15     FROM comps:c    //start at all vertices selected above
16     ORDER BY c.@localRelationshipCount DESC LIMIT 10;
17   PRINT comps[comps.@localRelationshipCount, comps.name];
18 }
```

Listing 4.60: A GSQL query that outputs the global number of WORK_AT relationships as well as the local number for the top 10 companies in that regard.

We ran all queries in this chapter on a TigerGraph Developer Edition 3.0 to ensure their correctness. The implementations of the SNB queries as well as the schema were taken from the TigerGraph GitHub repository[25]. We adapted some of them such that they work on our schema or use syntax V2, and again: a full implementation of all queries given here can be found on our GitHub repository.

4.4.1 Structure Independent

To show the difference between the two syntax versions in GSQL, we provide an implementation of Query 1 in both versions in Algorithm 4.61. When using the default syntax (V1), we cannot start directly with all vertices of a given type. To achieve this, we have to explicitly assign those vertices to a variable (line 3, vPersons) and use this variable

[24] That are all pre-defined types in our data model.

[25] https://github.com/tigergraph/ecosys/tree/ldbc/ldbc_benchmark/tigergraph

in the FROM clause. This variable can be omitted in a query using syntax V2 and we can directly start at all Persons (line 8).

```
1  CREATE QUERY query1(STRING firstName, STRING lastName) FOR GRAPH ldbc_snb
       [syntax v2] {
2    //using syntax v1:
3    vPersons = {Person.*};  //initialized with all vertices of type 'Person'
4    res = SELECT p FROM vPersons:p
5      WHERE p.firstName == firstName AND p.lastName == lastName;
6
7    //using syntax v2, we do not need vPersons:
8    res = SELECT p FROM Person:p  //can directly use vertex type Person in V2
9      WHERE p.firstName == firstName AND p.lastName == lastName;
10
11   PRINT res;  //regardless of syntax, print found vertex
12 }
```

Listing 4.61: Query 1 as a GSQL query. Note that we give this query once using syntax V1 (lines 3-5) and once using syntax V2 (lines 8-9).

Algorithm 4.62 shows the use of control flow primitives and how they can be integrated into a query. In this query, we do not know whether the given messageId represents the ID of a Comment or a Post. To make this distinction, we use the to_vertex_set(ID(s), type) function that fetches all vertices with the given ID(s) of the given type. The global accumulator seed holds the single ID and we test in line 8 whether this ID represents a Comment. In this case, we output the values of that vertex. Otherwise, we fetch the Post with the given ID, use another select-from-accum query to set either the content or the imageFile as messageContent and output this.

```
1  CREATE QUERY is4(STRING messageId) FOR GRAPH ldbc_snb {
2    SetAccum<STRING> @@seed;
3    SumAccum<STRING> @messageContent;
4
5    @@seed += messageId;
6    vComments = to_vertex_set(@@seed, "Comment"); //get set<vertex> from id(s)
7
8    IF vComments.size() > 0 THEN  //ID belongs to a Comment
9      PRINT vComments[vComments.creationDate AS messageCreationDate,
10         vComments.content AS messageContent];
11   ELSE                          //ID belongs to a Post
12     vPost = to_vertex_set(@@seed, "Post");
13     vPost = SELECT v FROM vPost:v
14       ACCUM CASE WHEN v.content != "" THEN v.@messageContent += v.content
15                 ELSE v.@messageContent += v.imageFile
16             END;
17     PRINT vPost[vPost.creationDate, vPost.@messageContent];
18   END;
19 }
```

Listing 4.62: Query 2 (IS4) as a GSQL query.

Unlike in Cypher and PGQL, GSQL's data model supports collections. We can therefore store the languages spoken by a person in a collection and calculate the average size of those collections via the query in Algorithm 4.63. Apart from the `count` function used here, GSQL provides a variety of mathematical, boolean, bit, string, date(time) and collection-related operators and functions.

```
1 AvgAccum @@avgLangs;
2 persons = SELECT p FROM Person:p
3   ACCUM @@avgLangs += count(p.speaks);
```

Listing 4.63: Parts of Query 3 as a GSQL query.

4.4.2 Pattern Matching

As briefly mentioned before, GSQL queries using syntax V1 can only traverse a single edge (1-hop queries) whereas in syntax V2 we can define more sophisticated, multi-hop paths and repetitions. Furthermore, we have to be careful which version we use as the syntax used to depict edges differs between them. Algorithm 4.64 depicts some of the differences. Edges can only be either undirected or right directed using ">" outside the () brackets in V1. In V2 however, edge directions are given next to the types inside the brackets and they can be either undirected, right (">") or left ("<") directed [26]. Having the directions inside the brackets allows for patterns in multiple directions as in line 3. Furthermore, syntax V2 allows us to match edges of any type as in line 5.

```
1 Person:p -(LIKES:e)-> Message:m  //syntax V1
2   //syntax V2:
3 Person:p -(KNOWS:e)- Person:m    //undirected edge
4 Person:p -((LIKES>|<HAS_CREATOR):e)- Message:m  //alternative types
5 Person:p -(_>:e)- Message:m      //any-type, right directed edge
```

Listing 4.64: Examples of path patterns in syntax V1 and V2.

We omit the implementations of Queries 4 and 5 as these contain no new functionality. The implementation of Query 6 in Algorithm 4.65 however shows multiple accumulators and a custom type and we will go over this in more detail. GSQL supports custom tuple types, that are containers holding a fixed sequence of base types as in line 2. These types are often used to aggregate related values. We use our custom `reply` type in a `HeapAccum` where we order the entries on the last name of the reply author (`replyLN`). The query is structured as follows: the values of the local `knows` accumulators are set to `True` for all persons that know the original author in lines 7-9. In lines 11-17, we first match all 1-hop comments and their authors and aggregate their information in an `ACCUM` into local accumulators (lines 13-16). The `CASE` statement in line 16 tests whether the reply author knows the original author and sets the `knows` value of the

[26] Note however that an edge type in GSQL is either declared directed or undirected and we can only match with the corresponding pattern: if the edge type E1 is declared undirected, we have to match it via `-(E1:e)-`, on directed edges we have to use a directed pattern `-(E1>:e)-` or `-(<E1:e)-` in V2. The same applies to V1, where we are however limited to right directed patterns `-(E1:e)->`.

Comment correspondingly. After these values are aggregated, we use POST-ACCUM in line 18 where we combine the values in a reply and add this to the global replyTop collection.

```
1  CREATE QUERY is7(STRING messageId) FOR GRAPH ldbc_snb syntax v2 {
2    TYPEDEF TUPLE<STRING content, STRING replyFN, STRING replyLN, BOOL
       knowEachOther> reply;  //custom type "reply"
3    OrAccum @knows;
4    SumAccum<STRING> @rAFN, @rALN;
5    HeapAccum<reply>(100, replyLN ASC) @@replyTop;  //aggregate results,
       ordered
6    //[...] convert messageId parameter to vertex vMessage
7    accFriend = SELECT s
8      FROM vMessage:s -(HAS_CREATOR>:e1)- Person:t1 -(KNOWS:e2)- Person:t2
9      ACCUM t2.@knows += True;  //set knows=True on all Persons that know t1
10
11   accReply = SELECT s
12     FROM vMessage:s -(<REPLY_OF:e1)- Comment:c -(HAS_CREATOR>:e2)- Person:t2
13     ACCUM
14       c.@rAFN = t2.firstName,
15       c.@rALN = t2.lastName,
16       CASE WHEN t2.@knows THEN c.@knows += True END //set knows-value
17     POST-ACCUM @@replyTop+= reply(c.content, c.@rAFN, c.@rALN, c.@knows);
18
19   PRINT @@replyTop;
20 }
```

Listing 4.65: Parts of Query 6 (IS7) as a GSQL query.

On a side note, TigerGraph added beta support for conjunctive patterns in version 3.0 of the database. These conjunctive patterns have to be correlated, meaning that at least one variable has to be shared among any two such patterns. Nonetheless, this allows complicated patterns to be decomposed into smaller ones and sophisticated patterns that cannot be given in a single pattern.

Evaluation Semantics

When matching graph patterns, GSQL uses a homomorphism-based bag semantics as default [DXWL19]. Therefore, multiple edges as well as vertices can be matched to the same variables (aliases). However, we have to keep in mind that a query in GSQL works different to queries in many other query languages as the SELECT clause can only take a single alias/variable as argument. As briefly mentioned above, we can store references to a vertex set in a variable, which is what we did in every query until now. In the query in Algorithm 4.66 for example, we store a reference to all selected Persons from the alias u in the variable per. However, we do not even need this variable later on as information is aggregated in the accumulators. When evaluating the pattern in line 4, GSQL will produce a match table where each row represents one distinct path, including ones where e=f and p=u [Tig]. We cannot restrict these matches but we can specify a list of aliases in the PER clause in order to group the entries in the match table on the values of those aliases. In our example, we list all aliases in that clause, which results

in one invocation of the ACCUM clause for every distinct (p,e,m,f,u) tuple, i.e. no grouping. Nonetheless, we can use PER to reduce invocations of the ACCUM clause which can change the perceived semantics as we often aggregate values in the latter clause. However, we cannot switch for example to an isomorphism-based semantics other than by specifying this requirement manually in the WHERE clause.

```
1 TYPEDEF TUPLE<UINT pid, DATETIME ecr, UINT mid, DATETIME fcr, UINT uid> info;
2 vPerson = {person};   //start person
3 per = SELECT u
4   FROM vPerson:p -(LIKES>:e)- (Post|Comment):m -(<LIKES:f)- Person:u
5   PER (p, e, m, f,u)  //not needed here as this is the default
6   ACCUM @@sum += 1,
7         @@pathInfos+= info(p.id, e.creationDate, m.id, f.creationDate, u.id);
8 PRINT @@sum, @@pathInfos;  //pathInfos is global HeapAccum<info>
```

Listing 4.66: Parts of a GSQL query in syntax V2 that aggregates path information.

4.4.3 Path

As mentioned before, the data model used by GSQL supports both, undirected and directed edges. Furthermore, each directed edge in the data model can have an inverse edge attached to them that is automatically synced with the original one, i.e. it will be created, updated and deleted automatically. Since all those edge types can appear in a single graph, the authors of GSQL extended 2RPQs and denote this extension as Direction-Aware Regular Path Expression (DARPE) [27]. Algorithm 4.67 contains some examples of the syntax of these DARPEs. As we can see in the pattern in line 1, repetitions of patterns are specified via "*" and we can provide lower and upper bounds similar to other graph query languages. On the other hand, there is no "+" for repetitions of at least one. However, this functionality can be achieved using a lower bound of 1 as in line 2. Line 3 contains a more sophisticated pattern and shows the dot operator that concatenates two edge patterns. This syntax simply removes the nodes between the edges and replaces them with a ".". Note however that all of this is only supported in syntax V2 as a query in syntax V1 can only ever traverse a single edge. [DXWL19]

```
1 Person:p -((LIKES>|<HAS_CREATOR)*3..5)- Message:m
2 Person:p -(LIKES*1..)- Message:m  //path of length at least 1
3 (Post|Comment):p -(<LIKES.IS_LOCATED_IN>.(IS_PART_OF>)*)- _:c  //dot operator
```

Listing 4.67: Examples of DARPEs in GSQL.

Algorithm 4.68 contains a small part of an implementation of Query 7. The full implementation is rather long as GSQL does not provide a function to get the shortest path between two vertices. However, there is an example of such an implementation on the TigerGraph website[28] and our implementation uses a similar approach. Generally, our

[27] Note that RPQs are also denoted as *Regular Path Expressions (RPEs)*. To this end, we can also think of DARPEs as DARPQs.

[28] https://docs.tigergraph.com/v/2.6/dev/gsql-examples/classic-graph-algorithms

query uses two Breadth-First Searches (BFS) starting at the two vertices (only one shown here), and stops as soon as a node that was already visited from the other BFS or the target node is found The expansion step in line 4 accesses the data graph and the rest of the code in this excerpt works solely on local and global accumulators. When we look at the code below, it looks more like it was written in a general programming language (`WHILE` loop, `IF/ELSE` statements, variable accesses) and not in a graph query language. However, this is exactly where we can see the expressiveness of the Turing-complete GSQL as we can essentially write any program in that language.

```
1  //[...]  query setup
2    WHILE NOT @@found DO
3      @@next = 0;
4      S1 = SELECT t FROM S1:s -(KNOWS)- Person:t WHERE t.@dist1 < 0
5        ACCUM
6          IF t.@dist2 > -1 THEN
7            @@found += True,   @@dist12 += s.@dist1 + t.@dist2 + 1
8          ELSE
9            @@next += 1,   t.@dist1 = s.@dist1 + 1
10         END;
11     IF @@found OR @@next == 0 THEN BREAK; END;
12     //[...] another BFS starting from the other person
```

Listing 4.68: Parts of Query 7 (IC13) as a GSQL query.

We omit the code for Query 8 as this query is again quite extensive and does not contain especially interesting concepts. Algorithm 4.69 contains an excerpt of our implementation of Query 9. We will not go into much detail and include it mainly for the path pattern in line 7. Apart from that, this query shows how we can linearly combine multiple queries. The result of the upper query is stored in the vertex set `vMessage` and used as starting point in the lower query (line 7).

```
1  //[...]
2  vPerson = { personId };
3  vMessage = SELECT t
4    FROM vPerson:s -(<HAS_CREATOR)- (Comment|Post):t
5    ORDER BY t.creationDate DESC, t.id DESC LIMIT 10;
6  accMessage = SELECT s
7    FROM vMessage:s -(REPLY_OF>*)- Post:t1 -(HAS_CREATOR>)- Person:t2
8    ACCUM //[...]
9  //[...]
```

Listing 4.69: Parts of Query 9 (IS2) as a GSQL query in syntax V2.

Evaluation Semantics

To avoid intractable performance when evaluating path queries, GSQL uses the all-shortest path semantics. This makes checking for the "*existence of a shortest path that satisfies a DARPE, and even counting all such shortest paths [...] tractable*" [DXWL19]. Regarding the output of a query, we have already mentioned that GSQL behaves different than many other query languages as we do not have to specify the output in the `SELECT` clause

but more often use accumulators. Nonetheless, we can PRINT and even RETURN single values, properties, but also entities and collections in accumulators. The functionality that is lacking compared to the other query languages that we analyzed is the possibility to output or to use paths in the query apart from the path patterns in the FROM clause.

4.4.4 Data Manipulation

GSQL is not only a DQL but also a DML, and both sub-languages are inspired by SQL [DXWL19]. This allows us to insert, update and delete elements in a graph, be it a vertex, an edge or a property. However, we can also see that the language is still under development as some approaches like the pattern matching syntax are not yet fully supported in DML queries. We refer the interested reader to the TigerGraph start guide about data modification[29] that lists these restrictions and how specific DML queries have to be structured. Nonetheless, we will go over the insertion and deletion constructs to show the general structure.

Algorithm 4.70 contains two INSERT INTO statements and we can see the resemblance to the counterpart in SQL. The first statement creates a vertex of type Person and sets all properties of that type. Note that types like Person are declared types and we have to set values for all declared properties. Edges can be inserted in ACCUM or POST-ACCUM clauses as in lines 2-4. In this example, we insert an edge of type STUDY_AT between the newly created person and the vertex u. Note that we ran into a problem when implementing the query as we did not manage to insert a variable number of edges and their properties in a single query. If we take the STUDY_AT edges for example, we would like to insert the classYear property (the empty attribute "_" in line 4) for each edge. However, if we structure this insertion as given below and the property values are given as BAG<INT> classYear, we do not know which of the entries in the bag belongs to the edge that is currently inserted. Though one could think that we can pass them in a bag of <ID, argument> tuples and iterate over these tuples, or even better in a map, GSQL queries can only take base types and sets/bags of base types as arguments. Base types in GSQL are integer as well as floating point numbers, booleans and strings, DATETIME (we use now() to get the current datetime) and also VERTEX and EDGE. We have more possibilities on local variables and output types as GSQL also supports MAP, JSONOBJECT and JSONARRAY in these cases. Nonetheless, we did not manage to implement the insertion of a variable number of edges where each edge has their own property value.

```
1 INSERT INTO Person VALUES (personId, firstName, lastName, gender, birthday,
    now(), locationIp, browserUsed, speaks, emails);
2 res = SELECT u FROM University:u WHERE u IN unis
3       ACCUM
4         INSERT INTO STUDY_AT VALUES(personId, u, _); //classYear empty: "_"
```

Listing 4.70: Parts of Query 10 (INS1) as a GSQL query.

[29] https://docs.tigergraph.com/v/3.0/start/gsql-102/adv/dml

Similar to Gremlin and PGQL, the removal of a vertex triggers the removal of all incident edges. Therefore, we can remove a comment, the incident edges and replying comments via the construct in Algorithm 4.71.

```
1 R = SELECT c FROM vComment:c -(<REPLY_OF*)- Comment:r
2     ACCUM DELETE(r)
3     POST-ACCUM DELETE(c);
```

Listing 4.71: Parts of Query 11 (DEL7) as a GSQL query (syntax V2).

Existing property values can be updated by setting a new value as in `SET e.classYear="2012"`. As GSQL uses a fixed schema, we cannot however add new or remove existing properties or change the labels of either vertices or edges in a DML query.

4.4.5 Data Definition

As mentioned in the introduction of this language, we have to declare the schema of a graph before we can interact with it. Using the `CREATE` statement of GSQL's DDL, we can create vertex and edge containers, the latter for directed as well as undirected edges. These containers define the type as well as the attributes/properties and their types. A graph is a named collection of these containers and containers can be shared by multiple graphs. In other words, we can specify which container types are available and contained in a graph. This allows for multiple views of the same data as we can for example create graphs `gA` and `gB` that each contain only some type containers and therefore instances of only those containers. Apart from multiple graphs that use the same containers and therefore offer different views of the same data, we can also create multiple graphs on completely unrelated containers and therefore on different data.

Access permissions for users on specific graphs together with the "`FOR GRAPH xyz`" statement that can be given on each query as in Algorithm 4.65 allow us to restrict or permit access to queries and therefore the underlying data to specific users [WD18]. In contrast to all the other query languages that we looked at, the data model does not include a dedicated label. Although the paper on the property graph type system in TigerGraph [WD18] mentions a label type, we could not find any reference to this in the current version of GSQL. However, we can use the names of the type containers similarly to labels in other query languages, `Comment:r` in Algorithm 4.71 for example restricts the match to instances of the `Comment` type container and resembles the functionality of labels.

Algorithm 4.72 shows an implementation of Query 13. In this example, we change the existing schema using a `SCHEMA_CHANGE` job that creates the vertex type `Street` and extends the definition of the `IS_LOCATED_IN` edge type. As the original definition of that edge type already includes `FROM(Person)` and `TO(City)`, all we have to add is the `Street` type as possible source and target. Note that running this job does not change any existing data and we have to adapt the existing relationships to match the

desired schema in a DML query. Apart from this version, we could also create a new graph that contains all existing vertex and edge container types and therefore the existing data. We can then create the `Street` type and a new edge type for the relationships to those vertices in the new graph. Therefore, we can have two versions using mostly the same data: the graph that already existed beforehand without streets and the new graph that also contains the streets.

```
1 USE GRAPH ldbc_snb  //specify the graph to work on
2 CREATE SCHEMA_CHANGE JOB add_street FOR GRAPH ldbc_snb {
3   CREATE VERTEX Street(PRIMARY_ID id UINT, name STRING, length INT);
4   ALTER EDGE IS_LOCATED_IN ADD FROM (Street) TO (Street)
5 }
6 RUN SCHEMA_CHANGE JOB add_street
```

Listing 4.72: Query 13 as a GSQL query.

On an interesting side note, the data model supports not only inverses of directed edges but also multiple parallel edges of the same type between the same vertices. In order to distinguish between these edges, we can specify a *discriminator attribute*, i.e. an attribute of the edge type that makes up the primary key of the edge together with the keys of the endpoints. [DXWL19]

4.4.6 Further Features

- **Composability:**
  Composability in GSQL queries works different to many other query languages. On the one hand, a GSQL query can contain multiple subqueries (select-from-where structures) as we used it in many of our examples. We can compose these subqueries linearly by saving the output of one selection (as a vertex set) and using this set in another subquery as starting point, as we did for example in line 7 of Algorithm 4.69. Apart from this, we can also use the imperative/procedural nature of GSQL where a query can either call another query or recursively itself. In that sense, a query is nothing else than a function and we can even return values using the `RETURN` statement. While we can use the output of such a query as input on another one, GSQL is not fully composable as this does not hold for all queries.

- **Paths as first-class citizens:**
  Unlike all other query languages that we looked at, we cannot even output paths let alone save them in a variable. Therefore, paths are not raised to first-class citizens.

Although GSQL tries to follow the syntax of SQL in many ways, it doesn't go without notice that the language focuses on achieving good performance on large graphs. To achieve this, GSQL uses a strong type system and requires the declaration of a schema that allows for better query optimization and potentially space savings. The language incorporates procedural concepts, is Turing-complete and therefore allows for complex

graph analytics workloads. As the TigerGraph database is developed as a parallel graph database, GSQL queries are designed in a way that allows for a computation in massively parallel processing (MPP) fashion [DXWL19]. All of this results in a syntax that is not as concise as in graph query languages like Cypher and PGQL [RH19].

Compared to other graph query languages, GSQL provides some interesting functions like `outdegree()` and `neighbors()` out of the box. Although these functions are often needed in OLAP workloads, they can also be useful in OLTP scenarios. Apart from the DQL, DML and DDL sub-languages, GSQL supports a Data Loading Language (DLL) that allows for the creation and execution of loading jobs. Again, these jobs are more likely of interest when using the database in an OLAP scenario to load for example the current, online data into an offline graph for analysis. However, this can just as well be used in OLTP scenarios to load data in the first place.

4.5 G-CORE

The lack of a standard query language and a uniform data model for interconnected data has led to the creation of a task force by the LDBC. This task force analyzed existing languages and identified three main challenges of graph querying. In 2016, members of the LDBC decided to design a new graph query language based on those challenges, namely: achieving full composability, treating paths as first-class citizens and capturing the core of available languages where possible [AAB+18]. The latter should ease the transition for users familiar with existing languages. Since Cypher was identified as the language that influenced the syntax of many graph query languages during the last decade, the new language G-CORE uses an ASCII-art syntax for patterns that is very similar to Cypher's. Another similarity to Cypher is that it is a high-level query language with a close alignment to SQL. Nonetheless, G-CORE does not only include features from SQL and Cypher but also from other languages like path patterns from PGQL. Unlike most graph query languages, G-CORE is not implemented or used in any database but is designed as a core for future languages that synthesizes desirable features. [Lin18]

To address to other two challenges, queries return a graph and the language uses an extension of the property graph data model called *path property graph (PPG)* that allows paths to be stored on their own. We introduced this model in Section 3.6 and revisit it in Section 4.5.4.

When we look at the query in Algorithm 4.73, we can see the resemblance between G-CORE and Cypher as both use mainly the same syntax on patterns, and SQL by the use of similar clauses, operators and functions. However, queries are not conceptually structured linearly as in Cypher but start similarly to SQL with the conceptually last clause that generates the result. As for the inner workings of G-CORE, the `MATCH` clause populates a binding table, where each binding is represented in a row that binds variables of the query to entities from the graph [AAB+18]. Note that we can optionally specify a graph name for each pattern in the `MATCH` clause, allowing for multiple patterns on different graphs in a single query. The `CONSTRUCT` clause then takes these bindings

and creates a graph from the values. In our example, the query returns a graph that is structured as specified in line 1, where all labels and properties from `p` and `f` are preserved and included in the resulting graph.

```
1 CONSTRUCT (p) -[:friend]-> (f)                          //return a graph
2 MATCH (p:Person) -[:KNOWS]- (f:Person) ON graph_name  //optional source
3 WHERE p.firstName = ? AND p.lastName = ?
```

Listing 4.73: A simple G-CORE query that returns a graph containing the `Person` with the given name and all persons they `KNOW`, and connects them via outgoing `friend` edges.

As the language is not implemented in any database, we use a parser that is available on GitHub[30] to ensure that the queries given here and provided in our GitHub repository are syntactically correct.

4.5.1 Structure Independent

```
1 CONSTRUCT (p)
2 MATCH (p:Person)
3 WHERE p.firstName = ? AND p.lastName = ?
```

Listing 4.74: Query 1 as a G-CORE query.

To achieve full composability, queries in G-CORE always return a graph [AAB+18]. The simple query in Algorithm 4.74 for example returns a graph that contains only a single vertex, but we can also return for example all `Persons` be removing line 3. The G-CORE paper however states that a potential implementation or an extension of the language can also use the `SELECT` clause that allows us to project from graphs or graph elements to tables. As the parser that we use to check our queries supports the select clause and we need it to implement many of our queries, we now assume that the language supports this clause. Note however that the `SELECT` clause violates the composability principle as such a query does not return a graph and the output can therefore not be used as input on another query. To this end, the paper mentions the possibility to include a `FROM` clause as an alternative to `MATCH` to also allow for tabular input data. However, the clause is only mentioned as a possible extension and we could not find any other reference. We do not need this clause in our examples but we have to keep in mind that outputs of `SELECT` queries potentially violate the composability principle.

The query in Algorithm 4.75 uses the `SELECT` clause and therefore produces a tabular output. As we can see in line 5, G-CORE supports the same compact notation as Cypher for simple restrictions on properties.

```
1 SELECT m.creationDate AS messageCreationDate,
2        CASE WHEN message.content IS NOT NULL
3             THEN message.content ELSE message.imageFile
```

[30] https://github.com/ldbc/ldbc_gcore_parser

```
4         END AS messageContent
5 MATCH (m:Message {id = ?})     //compact notation
```

Listing 4.75: Query 2 (IS4) as a G-CORE query.

Regarding the implementation of Query 3 we were constrained by the missing support for multi-valued properties or functions on them in some languages. While Cypher and Gremlin support such properties, we were only able to implement the query in a compact way in Cypher and not in a Gremlin traversal. PGQL's data model on the other hand does not support multi-valued properties at all, and in GSQL we were again able to implement this in a relatively compact way in using an `AvgAccum`. The data model of G-CORE supports such properties as well but it is not clear to us how to directly count the property values. To implement the query, we use a slightly different approach given in Algorithm 4.76. Caused by the assignment of the property `speaks` to the variable `s` in line 2, G-CORE automatically unrolls the values of the multi-value property `speaks`. As an example, assume that the property contains the two values `{"en","de"}` on the person with `id=123`. If the query did not contain the assignment, a single row in the binding table contains the information about that person, and all this information is contained in the single column `c` as this is the only variable apart from `s` in that query. Using the assignment however, the table will have the two columns `p` and `s` and we will have one row `[p=<person {id=123}>, s="de"]` and another one `[p=<person {id=123}>, s="en"]`. We then `GROUP` the entries on the person IDs which allows us to count the languages per person, and finally compute the average over those counts using `AVG`.

```
1 SELECT AVG( COUNT(s) ) as avgLanguages
2 MATCH (p:Person {speaks = s})  //unroll speaks into variable s
3 GROUP BY p.id
```

Listing 4.76: Query 3 as a G-CORE query.

According to the parser, G-CORE provides some functions from SQL like `SUM` and `MIN/MAX` on numbers, `COUNT` on elements as well as conditionals using `CASE` or inclusion checks via `IN`. We do not find any specifically graph relevant or otherwise new functions, most likely caused by the fact that it "*is intentionally designed as a small language that provides a kernel of graph matching and construction functionality*" [AAB+18].

4.5.2 Pattern Matching

G-CORE supports basic graph patterns in a syntax that is very similar to Cypher's as well as complex ones using the `UNION`, `INTERSECT` and `MINUS` set operations. As each query returns a graph, at least that's how it is conceived in the paper, i.e. without the `SELECT` operator, these set operations are applied to two graph queries that each return a graph and therefore work on graphs. Algorithm 4.77 shows an example of an implicit use of the UNION operator and of another property that follows from the closedness of the language, namely that views are possible. In this example, we assume that the

initial graph is named social_graph. The query creates a graph view that takes this graph and unions it with the construct in line 3. As this construct contains a pattern that already exists (the same pattern is matched in line 4), the query simply adds the property nr_messages to some knows edges in the initial graph. The union operation is implicit as all patterns and references given in the CONSTRUCT clause are merged together to create a single graph that is then returned. A possible result of this query can be seen in Figure 3.10 where the view adds the nr_messages properties whereas the rest already existed in the initial graph.

The query furthermore shows the OPTIONAL clause that can be used directly after a MATCH pattern or, as in this example, after the WHERE clause. We use this clause to optionally match a pattern over messages and their replies between the two Person nodes n and m. Having these patterns in our binding table allows us to count them and we set this value on the nr_messages property of the corresponding KNOWS edge (line 3). We also observe that graph patterns can not only occur in the MATCH and OPTIONAL but also in the WHERE clause. Whereas patterns in the other clauses are used to populate the binding table, patterns in the WHERE clause constrain the matches and therefore the entries in that table. Note that this is also possible in Cypher, where we did however not use it in any of our queries.

```
1 GRAPH VIEW social_graph1 AS (
2   CONSTRUCT social_graph,  //shorthand form for UNION operator
3         (n) -[e]-> (m) SET e.nr_messages := COUNT(*)
4   MATCH (n:Person) -[e:KNOWS]-> (m:Person)
5     WHERE (n) -[:IS_LOCATED_IN]-> () <-[:IS_LOCATED_IN]- (m)
6     OPTIONAL (n) <-[c1]- (msg1:Post|Comment),
7               (msg1) -[:REPLY_OF]- (msg2:Post|Comment) -[c2]-> (m)
8       WHERE (c1:HAS_CREATOR) AND (c2:HAS_CREATOR) )
```

Listing 4.77: A G-CORE query that creates a graph view. This is a slight variation of the query given in: G-CORE A Core for Future Graph Query Languages, page 7, lines 40-48 [AAB+18].

We skip Queries 4 and 5 as these closely resemble their implementations in Cypher and will briefly go over the query in Algorithm 4.78. This query uses a subquery to test whether the two authors know each other (line 2) and converts the result of the subquery to a boolean value using EXISTS. Although the definition of the PPG data model allows graph entities to be labeled with a set of labels, including the empty set, we follow the approach of LDBC's query examples and assume that all entities are only labeled with a single label. Nonetheless, G-CORE allows us to match on multiple labels as in line 4, similar to how this is possible in GSQL.

```
1 SELECT c.id, c.content, c.creationDate AS commentCreationDate,
2        p.id AS replyAuthorId, p.firstName, p.lastName,
3        EXISTS ( CONSTRUCT (m) MATCH (m) -[:HAS_CREATOR]-> (op:Person) -[r:
    KNOWS]- (p) ) AS replyAuthorKnowsOriginalMessageAuthor
4 MATCH (m:Post|Comment {id = ?}) <-[:REPLY_OF]- (c:Comment) -[:HAS_CREATOR]->
    (p:Person)
```

```
5 ORDER BY commentCreationDate DESC, replyAuthorId ASC
```

Listing 4.78: Simplified version of Query 6 (IS7) in G-CORE.

Evaluation Semantics
When evaluating a basic graph pattern, G-CORE uses the homomorphism-based semantics. Regarding the evaluation of complex graph patterns, we did not find a statement to whether a bag or set semantics is envisaged.

4.5.3 Path

Paths in G-CORE are demarcated with slashes `-//-` instead of `-[]-` and the language supports RPQs and by the support of navigational graph patterns also CRPQs [AAB+18, TKL19]. Furthermore, we can specify named path patterns that are very similar to path pattern macros in PGQL and we give an example later on in Algorithm 4.82. Using these path patterns, we can compare properties along a path which results in the support of a subset of REMs similar to PGQL.

The query in Algorithm 4.79 uses paths of variable length as well as the `SHORTEST` function that returns a single shortest path between the given nodes that satisfies the regular expression. We switch to reachability semantics by surrounding the regular- or named-path pattern with `<path_pattern>`. The existential subquery inside `EXISTS` in line 2 tests whether the nodes are connected over `knows` edges. If such a path was found, we derive the length of it via `length()`. Whereas the query in lines 1-6 returns a single number for the path length, the second version in lines 9-11 returns a graph that contains exactly that path.

```
1  SELECT
2     CASE WHEN EXISTS (CONSTRUCT () MATCH (p1)-/SHORTEST path <:knows*>/->(p2))
3          THEN (SELECT length(path) MATCH (p1)-/SHORTEST path <:knows*>/->(p2))
4          ELSE -1
5     END AS length
6  MATCH (p1:Person {id = ?}), (p2:Person {id = ?})
7
8  //another version that outputs a graph:
9  CONSTRUCT (p1) -/path/-> (p2)
10 MATCH (p1:Person) -/SHORTEST path <:knows*>/-> (p2:Person)
11 WHERE p1.id = ? AND p2.id = ?
```

Listing 4.79: Two versions of Query 7 (IC13) in G-CORE.

Compared to the implementations of Query 8 in Gremlin, PGQL and GSQL, where we included only a part of the queries because they were rather long, and even to Cypher's where we had to chain multiple query parts together, the implementation in G-CORE looks compact. As we can see in line 4 of Algorithm 4.80, reachability semantics and the default semantics can be mixed in a single pattern and we can give lower (`x`) and upper (`y`) bounds for the length of paths via `*{x..y}`. To construct the result, we start by selecting properties in the first line. In order to aggregate the study- and work-related

information, we use two subqueries in lines 2 and 3. Each of those queries contains a small pattern and multiple paths can be matched to those patterns when evaluating the query. We first collect the desired information from a single path in a string before these strings are aggregated using GROUP_CONCAT.

```
1 SELECT friend.id, friend.lastName, length(p) AS distance, friend.birthday,
      friend.creationDate, friend.gender, friend.browserUsed, friend.locationIP
      , friend.email, friend.speaks, friendCity.name,
2     ( SELECT GROUP_CONCAT(uni.name + ' ' + studyAt.year + ' ' + uc.name)
      MATCH (friend) -[studyAt:studyAt]-> (uni:University) -[:isLocatedIn]-> (
      uc:City) ) AS unis,
3     ( SELECT GROUP_CONCAT(company.name + ' ' + e.workFrom + ' ' + cc.name)
      MATCH (friend) -[e:worksAt]-> (company:Company) -[:isLocatedIn]-> (cc:
      Country) ) AS work
4 MATCH (person:Person) -/p <:knows*{1..3}> /-> (friend:Person) -[:isLocatedIn
      ]-> (friendCity:City)
5 WHERE person.id = ? AND friend.firstName = ? AND person <> friend
6 ORDER BY distance, friend.lastName, friend.id
```

Listing 4.80: Query 8 (IC1) as a G-CORE query.

We omit the implementation of Query 9 as it contains only a single navigational pattern.

Evaluation Semantics
When evaluating path queries, G-CORE uses a shortest-path semantics. Apart from querying for a single shortest path using SHORTEST, G-CORE supports "K SHORTEST" and even ALL to get all paths between two nodes. To avoid an infinite number of paths under ALL, this is limited to queries that return a subgraph and not the paths themselves. Regarding the output of a query, G-CORE provides more functionality than any other query language that we looked at as we can output everything from fixed arity outputs using SELECT to paths and even graphs using CONSTRUCT.

4.5.4 Data Manipulation & Data Definition

G-CORE does not support any statements to manipulate data. That includes not only the update operator but also the insertion and removal. Therefore, we cannot create a graph in the first place or manipulate an existing one using G-CORE as presented in [AAB+18]. However, it makes sense that the language includes only DQL functionality as it is meant as a proof of concept for future query languages and not for the direct use in a database. Therefore, the results of all our examples that use the CONSTRUCT clause are simply returned and not saved in a database. Furthermore, the SET operation that we us for example in Algorithm 4.77 does not change the existing data but the property in the result of that query alone.

As for the definition of a schema, G-CORE does not support it for the same reason. Regarding the data model, we have already mentioned that the language uses the PPG model. This is formally defined in [AAB+18] and allows for multiple named graphs. Edges in a graph have to be directed but we can query them in any direction. The major

addition in this data model is that paths are raised to first-class citizens. To that end, each graph has "*a (possibly empty) collection of paths; where a path is a concatenation of existing, adjacent, edges*" [AAB+18]. Moreover, each such path has its own identifier, can be labeled and even augmented with properties as we will see in Algorithm 4.82.

4.5.5 Further Features

- **Composability:**
  Developing a language that supports full composability was one of the main goals when designing G-CORE and is achieved with the `CONSTRUCT` clause that returns a graph. In order to also allow the output of paths while retaining closedness and therefore composability, paths have to be included in the data model which in turn leads to the PPG model. As mentioned before, the `SELECT` clause violates that principle as such a query no longer returns a graph but tabular data that cannot be used as input on another query. Assuming that we do not use the `SELECT` clause, we can naturally compose queries as the output of one query can be queried by another one. We can even use `SELECT` as long as a graph is specified in each `ON()`. Note that we omitted the `ON` clause in many examples as we assume a default graph in these cases. In Algorithm 4.81, we create a graph that consists only of vertices inside the `ON` clause (lines 3-7). This query uses another subquery to calculate the number of likes on a post and saves this value in the `likedBy` property. The outermost query then uses the graph constructed in lines 3-7 as input for the `MATCH` clause, matches these nodes, selects the top 10 on their `likedBy` value and returns that value together with the `title`. This example shows that it is possible to compose queries and we can include the `SELECT` clause in some scenarios. Nonetheless, we have to keep in mind that this query ultimately does not return a graph and therefore cannot be used as input on another one.

```
1  SELECT p.title AS title, p.likedBy as numLikes
2  MATCH (p) ON (
3       CONSTRUCT (p)
4         SET p.likedBy := (
5           SELECT COUNT(per) MATCH (per:Person) -[:likes]-> (p)
6         )
7       MATCH (p:Post) ON social_graph
8     )
9  ORDER BY numLikes DESC
10 LIMIT 10
```

Listing 4.81: A G-CORE query that demonstrates composability.

- **Paths as first-class citizens:**
  As mentioned earlier, paths are raised to first-class citizens in the PPG data model. To show this functionality, we provide an implementation of the query from Algorithm 4.50 in G-CORE and extend it such that the paths are stored and not only returned. This implementation is given in Algorithm 4.82. The query starts with

the definition of the path pattern `eq_voltage_hop`. In contrast to PGQL, slashes demarcate a path in G-CORE. This can be seen in line 3 where we match the 3 shortest paths between two nodes that satisfy the path pattern `eq_voltage_hop` and have a length of at least 1. As an extension to the implementation of the query in PGQL, we bind the `COST` to variable `c` such that we can use it later on in the `CONSTRUCT` clause. The default cost of a path is given by the hop-count which is why we can use this to get the length of a path. Apart from the default cost, users can also define their own cost function for paths. By specifying such a custom cost function, we can also query for the (top k) cheapest paths.

The `CONSTRUCT` clause in this query now contains one of the most important functions of G-CORE as we construct a graph with paths. We prefix the path variable `p` with `@` to denote that the path `p` is a stored path, i.e. it is not only returned in the graph consisting of vertices and edges but also on its own. To get a feeling on how such a result could look like, Figure 3.10 contains a graph as well as two stored paths[31]. We can see that the paths are stored apart from the vertices and edges. In our example, we label them by `eqVoltage` and add the `distance` property. Assuming that we already have a graph that contains such stored paths, we can query and analyze them, for example by accessing the nodes on the path via `nodes(path)`.

```
1  PATH eq_voltage_hop = (n:Device) -> (m:Device) WHERE n.voltage = m.
     voltage
2 CONSTRUCT (x) -/@p:eqVoltage {distance := c}/-> (y)
3  MATCH (x) -/3 SHORTEST p<~eq_voltage_hop*{1..}> COST c/-> (y)
4  WHERE x.name = 'generator_x29'
```

Listing 4.82: A similar query to the one in Algorithm 4.50. The query returns a graph with paths, i.e. a graph that contains not only vertices and edges but also paths.

It can be seen in many aspects that G-CORE was influenced by contemporary query languages and builds upon them. This results in the general approach using clauses similar to SQL, a pattern syntax that closely resembles Cypher's and named path patterns as in PGQL. Apart from the integration of existing concepts, the language shows that it is possible to achieve full composability and how paths, "*the most popular feature of graphs*" [AAB+18], can be raised to the same level as vertices and edges. All of this is achieved while remaining computationally feasible, meaning that each query can be evaluated in polynomial time with respect to the size of the data. Although we did not highlight and analyze the evaluation complexity in this thesis, it is interesting that the simple path semantics in Cypher can be NP-complete [FGG+18b] while the shortest path semantics in G-CORE allows the language to remain tractable, both with respect to the size of the data.

[31] Note that the graph in this figure has nothing to do with this query or even the domain of this query and the reference should solely help with the illustration of such a result.

4.6 Further Languages

We did analyze 5 contemporary graph query languages: Cypher, Gremlin, PGQL and GSQL as languages used in industry and implemented in database systems, and G-CORE as a core for future languages designed by industry and academia. Apart from these languages, there exists a wide range of further ones, both from academia and industry. An interesting example is BiQL [DNR09], a composable query language that does not distinguish between edges and nodes. The output of paths and a slight variation of the data model allows outputs to be stored and queried again, therefore achieving composability. What all of those languages lack however is standardization like SQL for relational databases or SPARQL for the RDF model. There are currently two major projects in standardization bodies that strive to achieve standardization for graph querying functionality and we will now introduce both of them.

/PGQ As briefly mentioned in Section 4.3, there is ongoing work inside the ISO/IEC working group responsible for SQL (SC32 WG3) to add graph querying functionality to SQL. This extension will be added in the next version of SQL, possibly SQL:2020 or SQL:2021 [TvR19]. As it is an extension to the relational query language, a tabular property graph data model where vertices and edges are both stored in SQL tables is envisioned. Tables can be mapped to named property graphs where table names become labels, columns properties and primary-foreign-key relationships edges (although all of that can be customized) using an extended DDL. If we look back on the DDL functionality in Section 4.3.5, this closely resembles PGQL's "`CREATE PROPERTY GRAPH`" statement. Indeed, examples of SQL/PGQ DDL statements in [TvR19] show a very similar syntax for the creation of property graphs. In contrast to PGQL however, the graph schema will be enforced as data is always stored in a relational database and the property graph provides solely a graph-view of that tabular data. We are therefore restricted by the data-types and structure of the underlying relational database.

Graphs are queried in a `MATCH` clause using an ASCII-art syntax similar to Cypher's that allows for fixed and variable length patterns as well as shortest path queries. This approach, where data is stored in relational tables and can be queried using either standard SQL or in a graph view, entails that data manipulation through standard SQL DML statements are automatically visible on graphs over that data. Although there is no final specification of this extension available yet, adding property graph functionality to SQL brings the world of graphs, patterns and paths to the large group of SQL developers. Furthermore, as this functionality will be standardized with SQL, the graph related parts can be influential for the development of other graph query languages.

GQL The same working group that is responsible for SQL and working on SQL/PGQ is also developing a new, independent property graph query language. As briefly mentioned in Section 2.4, this language is called GQL and is set to become the

standardized graph query language. Similar to SQL/PGQ, there is no specification for this available as of now, but a paper on the scope and features [GFL+18] proposes an initial design of the language and its features. Generally, GQL "*is a composable declarative database language for querying and managing property graphs that is intended to be usable both as an independent language as well as in conjunction with SQL or other languages*" [GFL+18]. During query evaluation, a driving table similar to the binding table in Cypher and G-CORE holds data. A GQL query either returns a graph created from these values, similar to `CONSTRUCT` in G-CORE, or returns data directly in tabular form as SQL's `SELECT`. In contrast to G-CORE, GQL is closed under graphs and tables, meaning that a query can take a graph or a table as input and output either of them[32]. This allows GQL to be used in conjunction with SQL and potentially other languages, and entails that it is composable.

Another interesting feature regards the supported evaluation semantics. As for patterns, the language strives to allow users to switch between homomorphism-based, no-repeated node and no-repeated edge semantics. Regarding the evaluation of paths, reachability, shortest, cheapest and their top K variants, all paths and even simple (no-repeated-nodes), trail (no-repeated edge) and acyclic path semantics should be supported. If the language indeed supports all of these semantics, it will essentially include all versions that we identified in Chapter 3.

Coming back to Figure 4.1, we can see that the language supports complex path patterns similar to the ones in PGQL (and therefore G-CORE) as well as an optional schema that is influenced by GSQL. The figure contains not only GQL and existing graph query languages but also SQL/PGQ. This is especially interesting as GQL and SQL/PGQ are developed by the same working group and indeed, common core features will be specified in one standard and referenced by the other [HLP+19]. The dependencies between the two projects can be seen in Figure 4.2 where `Read GQL` for example specifies graph related functions like pattern matching that are needed in both projects and `GQL Proper` includes functions that are only supported by GQL.

Although currently available documents on GQL promise a feature-rich graph query language with compatibility to SQL, it remains to be seen how much of this functionality is included in the final specification. And, as briefly mentioned before, the specification alone is not enough for it to be accepted and implemented in graph databases and only time will tell if, and how fast, this happens. Nonetheless, we are today closer to a world where a standardized language is broadly supported in many graph databases than we were some 5 years ago, before the first initiatives towards standardization.

[32] More specifically, possible in- and outputs include not only graphs and tables but also singular values or nothing.

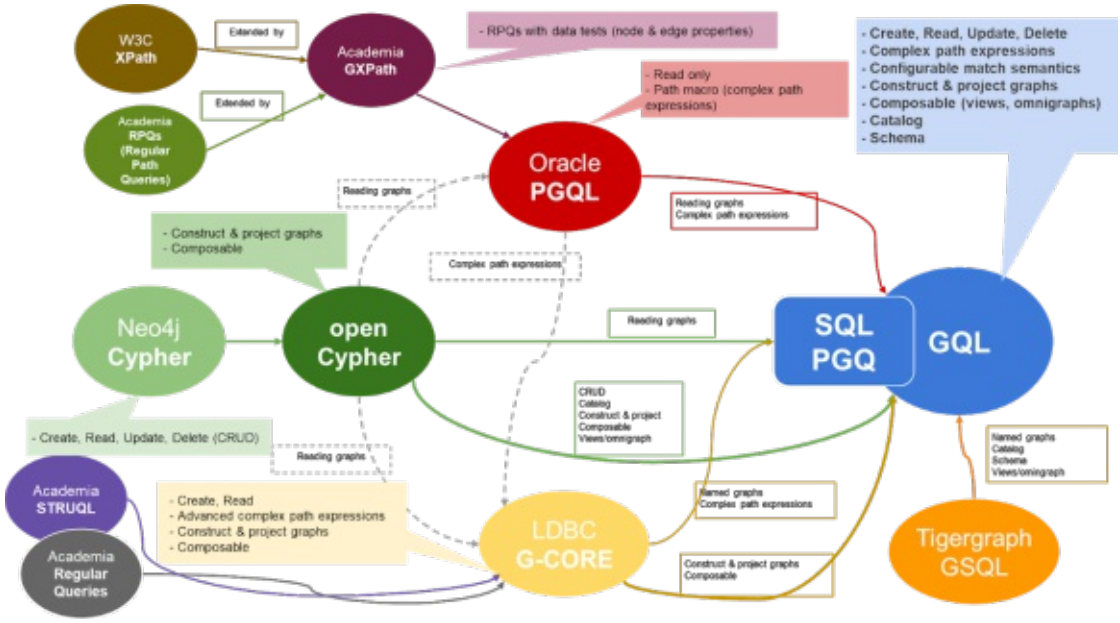


Figure 4.1: GQL Lineage. Source: GQL Standards - Existing Languages, Petra Selmer [GQL].

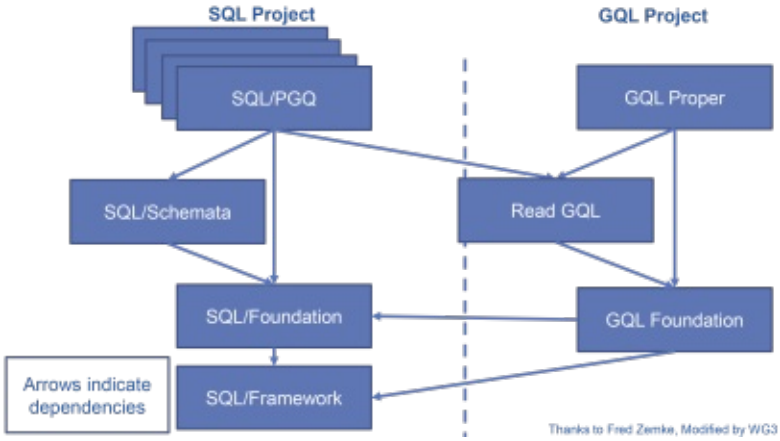


Figure 4.2: Dependencies between SQL and GQL Projects. Source: SQL and GQL, slide 17 [HLP+19].

4.7 Lessons Learned

Now that we analyzed five graph query languages, we wrap this chapter up by comparing them on their support for certain features, their evaluation semantics and other general characteristics. We also include guidelines on which languages fit which application scenario and for whom it makes sense to choose a certain language. Table 4.1 provides an overview of the support for certain graph related features in the analyzed languages. The upper half of the tables compares functionality related to the data model, like the support for multiple graphs, undirected edges or multi-value properties. The lower half then focuses on querying related features like the support of branches (e.g. via the `CASE` statement), a linear or nested composition of queries and whether users can provide and execute their own function. An overview of the supported evaluation semantics for the evaluation of patterns and paths can be seen in Table 4.2.

We now go over the analyzed languages one more time, highlight their characteristics, advantages and disadvantages and give examples for when to use them. Each section furthermore includes an explanation about what is missing towards a full implementation of the queries from Chapter 3 that are only partially or not at all implemented. Table 4.3 contains an overview about the query implementation status.

Cypher
Cypher is a high-level declarative language that focuses heavily on patterns, provides many graph related functions out of the box and allows for extensions via user-defined functions. Being "*the original declarative query language for the property graph data model*" [Pla18], the ASCII-art syntax for patterns pioneered by Cypher influenced other contemporary graph query languages like PGQL and G-CORE. Cypher was first introduced in Neo4j 1.4 in 2011 [Lin18] and is therefore one of the older contemporary graph query languages, also compared to the other languages that we analyzed. The development and governance of the language by the openCypher Project allows other companies and bodies to implement the language in their systems. Together with the fact that Neo4j is one of the most popular property graph databases and Cypher therefore relatively well known in the community, this led to the inclusion of Cypher in products like SAP HANA Graph, Redis Graph and Memgraph [FGG+18b]. Furthermore, Cypher uses a concise syntax and aligns relatively closely with SQL, which eases the transition for users familiar with the relational query language. All of this leads to a large community around the language, which in turn results in a multitude of forums, tutorials and guides as well as user-defined functions that are available on the internet.

Regarding the queries introduced in Chapter 3 we were able to implement all except the last one that is DDL related as Cypher does not support a schema. The support of a multitude of graph-specific functions like shortest path finding or matching optional patterns allows for (subjectively) understandable queries that are relatively easy to write, even for non-experts in the language. Caused by the focus on patterns and the rich set of features (not only DQL- but also DML-related), Cypher allows for expressive graph queries and can be used in many graph related scenarios. In other words, Cypher

Table 4.1: Support for certain features in the analyzed graph query languages.

Feature	Cypher 9	Gremlin	PGQL	GSQL	G-CORE	Additional Remarks
data model						
multiple graphs	✗(Neo4j: ✓)	✓	✓	✓	✓	
undirected edges (data)	✗	✗	✗	✓	✗	
node labels	0-n	1	0-n	1*	0-n	*No labels, but we can use the declared type.
edge labels	1	1	0-n	1*	0-n	*See remark above.
add/remove label	(✗| ✓)*	✗	✗	✗	✗	*The single edge label cannot be changed but node label(s) can.
add/remove property	✓	✓	✓	✗	✗*	*In result: yes, but no DML functionality.
multi-value properties	✓	✓	✗	✓	✓	
paths first-class citizens	✗	✗	✗	✗	✓	
querying						
path querying class	⊂ RPQ	⊃ RPQ	⊂ REM	DARPE	⊂ REM	
ASCII-art pattern	✓	✗	✓	✓	✓	
(named) path patterns	✲	✲	✓	✗	✓	✲ Can assign a matched path to variable/alias, but not a generic, yet unmatched pattern.
branches	✓	✓	✓	✓	✓	
loops	✗	✓	✗	✓	✗	
linear composition	✓	✗*	✗	✓	✓	*In our version (TinkerPop 3.0.1). ✓in newer versions.
nested subqueries	✗(v.10: ✓)	✓	✓	✓	✓	
composable language	✗(v.10: ✓)	✗	✗	✗	(✗)*	* In the sense that a query can return a vertex (set of vertices), that can be used as input. Does not hold generally.
user-defined functions	✓	✓*	✓	✓	✗	*In DSL or directly in programming language.
DML	✓	✓	✓	✓	✗	
DDL	✗*	✗	✗	✓	✓	*In Neo4j: some constraints possible.

	homo-morphism	isomorphism based				
		n-r-edge	n-r-node	n-r-anything	shortest	reachability
Cypher		♦ ✣				
Gremlin	♦ ✣		(✣)			
PGQL	♦	(♦)	(♦)	(♦)	(✣)	(✣)
GSQL	♦				✣	
G-CORE	♦				✣	✣

Table 4.2: Evaluation Semantics used by the analyzed query languages. n-r-x stands for no-repeated-x. A ♦ denotes that this semantics is used by default to match patterns whereas ✣ denotes the default semantics when matching paths. For paths in PGQL: variable length path patterns can only appear in reachability semantics (using `-//-`) or inside `SHORTEST` for the shortest path semantics. In G-CORE, shortest path semantics is used by default but path expressions demarcated with `-/<>/-` are evaluated using reachability semantics. (♦) and (✣) denotes that we can change to this semantics, in many cases by adding a dedicated statement like `simplePath()` in Gremlin to the query.

	fully implemented	**partially implemented**	**not doable**
Cypher	1 - 12		13: DDL
Gremlin	1 - 6, 8 - 12	7: persons not connected ★	13: DDL
PGQL	1, 2, 4 - 9, 11, 12	3: multi-value-property / list 10: variable number of edges 13: enforced schema	
GSQL	1 - 9, 11 - 13	10: variable number of edges	
G-CORE	1 - 9	10 - 13: DML & DDL	

Table 4.3: Implementation status of the queries from Chapter 3. The text after some query numbers states what is missing for a (full) implementation. ★ Our implementation works only if the two `person` vertices are connected over `knows` edges. In other words: Gremlin lacks a functionality to test whether two vertices are connected.

is a solid choice for any application scenario as long as a schema or more expressive path patterns are not needed. Furthermore, it is a good idea to use Cypher for its close alignment to SQL and the large community when first starting with a graph query language. Nonetheless, one has to keep in mind that Cypher for example does not support full RPQs let alone more expressive path patterns, currently lacks support for a schema or other constraints and for more sophisticated features like composability.

Gremlin

In contrast to Cypher, Gremlin is more imperative in nature. It uses a completely different approach to all the other languages that we analyzed as it merges with a programming language that implements the querying functionality. This allows queries to not only

use graph traversal functionality but also constructs of the implementing programming language like general loops. Furthermore, users can achieve a shorter syntax tailored to their domain using a Domain Specific Language (DSL) and Gremlin can be easily extended inside the implementing programming language. For all of that, Gremlin is Turing-complete [WD18] and allows for a superset of RPQs as we can repeat whole traversals [AAB+18]. Nonetheless, being a low-level query language with a focus on the imperative traversal-style [Rod15] together with the compact syntax leads to worse readability compared to higher-level languages [HP13]. We would therefore not advise developers that are familiar with declarative programming- or query languages like SQL to switch directly to Gremlin. Changing to, or choosing Gremlin for a project makes more sense for someone familiar with functional languages as a general traversal step is nothing but a function and the language heavily relies on function composition and nesting.

Apart from the query language, the TinkerPop framework and GTM provide a distributed mode where queries can be executed on distributed machines. Together with a dedicated OLAP API, this results in a viable alternative for big data scenarios. Therefore, Gremlin is a reasonable choice for someone who is not only interested in OLTP workloads but also in the analysis of larger amounts of interconnected data.

We managed to implement most of the analyzed queries except for two: we lack a functionality to test whether a pair of vertices is connected for a full implementation of Query 7 and the general support of a schema for Query 13. Other than that, Gremlin provides many interesting graph related functions out of the box and uses a rich data model that allows not only for multiple graphs but also for meta-properties.

PGQL

PGQL is again a high-level declarative query language that was influenced by Cypher and SQL [Pla18]. This results in a syntax similar to Cypher's and a close alignment to SQL for the use of the same functions and mostly the same data types. Up until the release of the current version of PGQL in March 2020, the language lacked important functionality like a DML sub-language [Ora20]. With this version however, PGQL provides a solid core of graph related functions. Compared to Cypher and Gremlin, the inclusion of path pattern macros allows not only for full support of RPQs but even for the comparison of properties on a path and therefore a subset of REMs [vR17]. Apart from path patterns, Cypher and Gremlin still provide more graph related functions out of the box. Unlike Cypher that is implemented in multiple (commercial) products, PGQL is only available in some Oracle products where data is often stored in relational databases. This comes with an advantage as well as a disadvantage. On the one hand, including PGQL in the Oracle Database allows the large group of existing users to utilize graph querying on their already available data in a relational database. On the other hand, the language is still comparatively new and as PGQL is not utilized by any other vendor, there is only a rather small community and we could not find much information apart from the official specification.

It is a rational choice to look into PGQL for someone who is already using an Oracle

product. Nonetheless, one has to be careful when choosing a product for their use case, as PGQL in the Oracle Database will result in a graph view of relational data whereas PGQL in "Oracle Big Data Spatial and Graph" can utilize PGX where data can be stored in a native graph database. We did not go into detail regarding the performance and thus did not test this but it might become problematic when data is stored in tables, especially when matching path patterns that likely result in a multitude of relational joins. If we restrict ourselves to the query language alone, many arguments for Cypher, like the alignment to SQL and the ASCII-art syntax that results in concise and (subjectively) understandable queries, apply to PGQL as well.

As for the analyzed queries, we were able to fully implement 10 out of the 13. Lacking support for multi-value properties or lists hindered us towards a full implementation of Query 3. Note that we are not able to store a collection of values (the spoken languages) in a single property in PGQL's data model whereas this is possible in the other data models that we analyzed. We lack the ability to pass tuples or iterate over collections inside a query for a concise implementation of Query 10. Note however that the desired functionality can be achieved by splitting the query into multiple ones. Regarding Query 13, we can use the "`CREATE PROPERTY GRAPH`" statement to create a graph with the desired structure out of tabular data, but this schema is only used to create the graph and not enforced later on.

GSQL

Similar to PGQL, GSQL is a high-level query language inspired by SQL. It is used in the TigerGraph database and developed by the homonymous company. The most interesting aspect of GSQL is the use of a strong type system and schema. This type system extends the one from SQL by graph specific types like `vertex`, `edge` and `graph` [DXWL19]. Graphs include certain vertex and edge types that specify exactly what entities are included/allowed in the graph and how they can be related. The schema allows for better query optimization and storage savings, which is especially needed in TigerGraph as it is designed for "*tomorrow's big data and analytics*" [HLP+19] and therefore for huge graphs. This explains the distributed approach with native support for parallel query execution and Accumulators. Since this resembles the MapReduce model, GSQL is a reasonable choice for someone with a background on this or a similar NoSQL paradigm [DXWL19]. Having a schema is not the only argument for GSQL as the language provides not only declarative constructs but also imperative ones. A single GSQL query can be composed of multiple select-from-where structures together with imperative constructs like loops or branches. This results in GSQL being Turing-complete. As for path querying capabilities, the language supports not only RPQs but the more expressive class of DARPEs that allows for directed as well as undirected edges in a single graph.

Although the language provides OLTP functionality, it is targeted at OLAP workloads. Nonetheless, GSQL is an expressive query language that can be used for OLTP workloads and is especially interesting when dealing with large amounts of interconnected data. On the other hand, the schema, the inclusion of imperative constructs and the parallel approach using accumulators results in a syntax that is not as concise as Cypher's [RH19].

The language furthermore lacks some built-in functionality provided by other languages, like a function to get the shortest path between two vertices. Taken together, this results in queries that are often more complex, not only to develop but also to read and understand as they generally contain more code. This augmented code is either needed to make up for nonexistent built-in functions or for the distributed nature. Although the language is only implemented in the TigerGraph database, there are many resources and tutorials on their website and an active community in some forums.

Regarding the analyzed queries, we managed to fully implement all of them except for Query 10. In contrast to PGQL, we are able to insert a variable number of edges by passing a collection of edge endpoints. However, we did not manage to correctly set a different property for each edge when adding a variable number of edges. Note again that we can achieve this functionality by splitting the query into multiple ones.

G-CORE

One goal when designing the research language was to capture the core of available languages. Since G-CORE is a high-level declarative language, it is not surprising that it uses an ASCII-art syntax very similar to Cypher's and incorporates the more expressive path patterns from PGQL. The language is designed as a core for future developments and shows how a sophisticated graph query language that integrates industrial experiences with theoretical research can look like [AAB+18]. As for the inclusion of new functionality, the language is fully composable and raises paths to first-class citizens by extending the property graph data model to also include paths. This results in a powerful query language while retaining a syntax that closely resembles Cypher's. Although this makes the language relatively easy to use and understand while extending its expressiveness, it does not make sense to recommend using G-CORE as it is not implemented in any database system.

We were able to implement all analyzed DQL queries in G-CORE but no DML or DDL query as the language is solely a query language without support for data manipulation or a schema.

For convenience, we note again the versions of the languages that we analyzed. Regarding Cypher, we analyzed Version 9 of the language from the openCypher project as well as some differences to the version of Cypher as implemented in the Neo4j 4.1.0 database. With version 10 of the language being currently under development and said to add interesting functionality like composability, we also mentioned potential functionality and concepts of this upcoming version. Regarding Gremlin, we analyzed a slightly older version of the TinkerPop framework and therefore Gremlin (3.0.1-incubating) but mentioned notable differences to more recent ones. For PGQL, we analyzed the newest version 1.3 and for GSQL we used the version that comes with TigerGraph 3.0, which is currently the latest stable version of that database.

CHAPTER 5

Conclusion

Graph databases are increasingly adopted in industry to handle, store and analyze graph-like data. Users can choose between a multitude of graph databases, but there is no standardized query language to access them and most systems use their own data model. Although work on the standardized graph query language GQL started in 2019, it is set to take multiple years until it is finished and standardized. As current graph query languages differ in their characteristics, expressiveness and ease of use, it is important to choose a language that fits your needs. We therefore analyzed five common graph query languages regarding these factors.

In this thesis, we analyzed common graph query languages and produced the following results:

- We identified and analyzed common features that form the conceptual core of most graph query languages by inspecting queries from the Social Network Benchmark. Matching graph patterns is a, if not the, major operation on graphs. These patterns can be extended with path expressions to form navigational queries. We added three more features: structure independent, data manipulation and data definition. The features are described and analyzed regarding their theoretical foundations and the expressiveness they enable.
- In addition, we analyzed four of the most prominent graph query languages from industry (Cypher, Gremlin, PGQL, GSQL) and one promising candidate for future languages from a research project (G-CORE) regarding these features. Cypher offers a multitude of graph-related functions and is relatively easy to use. Although Gremlin is more expressive (Turing-complete), its concise syntax and imperative nature leads to worse readability compared to higher-level languages. PGQL offers expressive path patterns but does not provide as many graph-related functions as Cypher. GSQL is the only language in our comparison with a type system and a

schema, provides not only declarative querying functionality but also imperative constructs and is Turing-complete. We conclude with G-CORE, an expressive language that combines features mostly from Cypher and PGQL, achieves full composability and elevates paths to first-class citizens in the graph.

We limited our comparison mostly to the expressiveness and ease of use of the analyzed languages. It would therefore be naturally of interest to also analyze the complexity of the languages and compare their performance on certain workloads. Although there are some papers that focus on these matters, like [HP16] for the performance of Cypher and Gremlin or [AAB+17] for an analysis of the same languages regarding their complexity, there is little on PGQL and GSQL. As an example, it would be interesting to compare Gremlin with GSQL as both offer a native parallel mode and imperative constructs. Similarly, it would be interesting to compare PGQL with Cypher (and G-CORE) as they provide a similar syntax but differ in the evaluation semantics and the data representation if we pick Neo4j and the Oracle database for example.

List of Figures

List of Tables

Listings

Acronyms

2RPQ Two-way Regular Path Query. 30, 32, 90

CRPQ Conjunctive Regular Path Query. 31, 32, 99

DARPE Direction-Aware Regular Path Expression. 90, 107, 110

DDL Data Definition Language. 11, 17, 37, 41, 93, 95, 103, 106–108, 111

DML Data Manipulation Language. 11, 17, 37, 38, 41, 47, 55, 81, 92–95, 103, 106–109, 111

DQL Data Query Language. 11, 37, 47, 80, 92, 95, 100, 106, 111

ECRPQ Extended Conjunctive Regular Path Query. 32, 33, 42, 78

GQL Graph Query Language. 13, 74, 103, 113

GTM Gremlin Traversal Machine. 15, 59, 62, 109

ISO International Organization for Standardization. 13, 74, 103

LDBC Linked Data Benchmark Council. 16, 17, 22, 27, 35, 42, 62, 95, 98

OLAP Online Analytical Processing. 4, 15, 17, 84, 95, 109, 110

OLTP Online Transactional Processing. 4, 15–17, 73, 74, 84, 95, 109, 110

PGX Parallel Graph AnalytiX. 15, 16, 81, 110

PPG Path Property Graph. 42, 95, 98, 100, 101

RDF Resource Description Framework. 8, 40, 103

REM Regular Expressions with Memory. 32, 78, 99, 107, 109

RPQ Regular Path Query. 29–36, 52, 68, 78, 90, 99, 107–110

SNB Social Network Benchmark. 17–19, 22, 27, 35, 38, 40, 41, 45, 66, 78, 86, 113, 115

SQL Structured Query Language. 1, 6, 11–16, 22, 42, 46–48, 51, 63, 74, 76–78, 81–84, 92, 94, 95, 97, 102, 103, 106, 109, 110

W3C World Wide Web Consortium. 8, 9, 13

Bibliography

[AAA+20] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The LDBC social network benchmark. *CoRR*, abs/2001.02299, 2020. [current stable version: 0.3.2].

[AAB+17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

[AAB+18] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432. ACM, 2018.

[ABR13] Renzo Angles, Pablo Barceló, and Gonzalo Rios. A practical query language for graph dbs. In Loreto Bravo and Maurizio Lenzerini, editors, *Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management, Puebla/Cholula, Mexico, May 21-23, 2013*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[ACP12] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In Alain Mille, Fabien L. Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab, editors, *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 629–638. ACM, 2012.

[AG08] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1:1–1:39, 2008.

[AG18] Renzo Angles and Claudio Gutierrez. An introduction to graph data management. In George H. L. Fletcher, Jan Hidders, and Josep-Lluís Larriba-Pey, editors, *Graph Data Management, Fundamental Issues and Recent Developments*, Data-Centric Systems and Applications, pages 1–32. Springer, 2018.

[Ang12] Renzo Angles. A comparison of current graph database models. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, pages 171–177. IEEE Computer Society, 2012.

[Ang18] Renzo Angles. The property graph database model. In Dan Olteanu and Barbara Poblete, editors, *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[ARV19] Renzo Angles, Juan L. Reutter, and Hannes Voigt. Graph query languages. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

[AS92] Bernd Amann and Michel Scholl. Gram: A graph data model and query language. In Dario Lucarella, Jocelyne Nanard, Marc Nanard, and Paolo Paolini, editors, *ECHT '92: European Conference on Hypertext Technology, November 30 - December 4, 1992, Milan, Italy*, pages 201–211. ACM, 1992.

[Bar13] Pablo Barceló. Querying graph databases. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 175–188. ACM, 2013.

[BFL12] Pablo Barceló, Diego Figueira, and Leonid Libkin. Graph logics with rational relations and the generalized intersection problem. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 115–124. IEEE Computer Society, 2012.

[BLLW12] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems*, 37(4):31:1–31:46, 2012.

[BLR11] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. Querying graph patterns. In Maurizio Lenzerini and Thomas Schwentick, editors, *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 199–210. ACM, 2011.

[BPG+19] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *CoRR*, abs/1910.09017, 2019.

[BPR12] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. Relative expressiveness of nested regular expressions. In Juliana Freire and Dan Suciu, editors, *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*, volume 866 of *CEUR Workshop Proceedings*, pages 180–195. CEUR-WS.org, 2012.

[CB05] Thomas M. Connolly and Carolyn E. Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.

[CGLV00a] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000*, pages 176–185. Morgan Kaufmann, 2000.

[CGLV00b] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Query processing using views for regular path queries with inverse. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2000*, pages 58–66, 2000.

[CGLV03] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[Cha12] Mark Chatham. *Structured Query Language By Example-Volume I: Data Query Language*. Lulu.com, 2012.

[CL14] Arnaud Castelltort and Anne Laurent. Fuzzy queries over nosql graph databases: Perspectives for extending the cypher language. In Anne Laurent, Olivier Strauss, Bernadette Bouchon-Meunier, and Ronald R. Yager, editors, *Information Processing and Management of Uncertainty in Knowledge-Based Systems - 15th International Conference, IPMU 2014, Montpellier, France, July 15-19, 2014, Proceedings, Part III*, volume 444 of *Communications in Computer and Information Science*, pages 384–395. Springer, 2014.

[CM90] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In Daniel J. Rosenkrantz and Yehoshua Sagiv, editors, *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*, pages 404–416. ACM Press, 1990.

[CMW87] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, USA, May 27-29, 1987*, pages 323–330. ACM Press, 1987.

[CMW88] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. G+: recursive queries without recursion. In Larry Kerschberg, editor, *Expert Database Systems, Proceedings from the Second International Conference, Vienna, Virginia, USA, April 25-27, 1988*, pages 645–666. Benjamin/Cummings, 1988.

[Cod80] E. F. Codd. Data models in database management. In Michael L. Brodie and Stephen N. Zilles, editors, *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, Pingree Park, Colorado, USA, June 23-26, 1980*, volume 11, pages 112–114. ACM Press, 1980.

[DNR09] Anton Dries, Siegfried Nijssen, and Luc De Raedt. A query language for analyzing networks. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy J. Lin, editors, *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 485–494. ACM, 2009.

[DWX18] Alin Deutsch, Mingxi Wu, and Yu Xu. Seamless syntactic and semantic integration of query primitives over relational and graph data in GSQL. `https://cdn2.hubspot.net/hubfs/4114546/Collateral/IntegrationQuery%20PrimitivesGSQL.pdf`, 2018. [Online; last accessed November 2020].

[DXWL19] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019.

[EAL+15] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC social network benchmark: Interactive workload. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 619–630. ACM, 2015.

[FB18] Diogo Fernandes and Jorge Bernardino. Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In Jorge Bernardino and Christoph Quix, editors, *Proceedings of the 7th International Conference on Data Science, Technology and Applications, DATA 2018, Porto, Portugal, July 26-28, 2018*, pages 373–380. SciTePress, 2018.

[FGG+18a] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. Formal semantics of the language cypher. *CoRR*, abs/1802.09984, 2018.

[FGG+18b] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018.

[FLS98] Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 139–148. ACM Press, 1998.

[FPSW19] George H. L. Fletcher, Alexandra Poulovassilis, Petra Selmer, and Peter T. Wood. Approximate querying for the property graph language cypher. In *2019 IEEE International Conference on Big Data, Los Angeles, CA, USA, December 9-12, 2019*, pages 617–622. IEEE, 2019.

[GFL+18] Alastair Green, Peter Furniss, Tobias Lindaaker, Petra Selmer, Hannes Voigt, and Stefan Plantikow. GQL Scope and features. `https://s3.amazonaws.com/artifacts.opencypher.org/website/materials/sql-pg-2018-0046r3-GQL-Scope-and-Features.pdf`, alternative: `https://drive.google.com/file/d/1nQXzl7u5eU0iu8s1Q7ZdBSLuPGBxjZWF/view`, December 2018. [Online; last accessed November 2020; linked from `http://www.opencypher.org/references`].

[GJK+18] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. opencypher: New directions in property graph querying. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 520–523. OpenProceedings.org, 2018.

[GQL] Existing languages. `https://www.gqlstandards.org/existing-languages`. [Online; last accessed November 2020].

[Gre18] Alastair Green. The GQL manifesto - one property graph query language. `https://gql.today`, May 2018. [Online; last accessed May 2020].

[Gre19a] Alastair Green. GQL is now a global standards project alongside SQL. https://neo4j.com/blog/gql-standard-query-language-property-graphs, September 2019. [Online; last accessed May 2020].

[Gre19b] Alastair Green. Graph query language GQL - critical milestone for ISO graph query standard GQL. https://www.gqlstandards.org/gql-blogs/critical-milestone-for-iso-graph-query-standard-gql, June 2019. [Online; last accessed May 2020].

[HLP+19] Keith Hare, Victor Lee, Stefan Plantikow, Oskar van Rest, and Jan Michels. SQL and GQL. https://drive.google.com/file/d/1hwmJt9CqD8C8zoFskGnZ44eX2MZooyHM/view, March 2019. [Online; last accessed May 2020; linked from https://www.gqlstandards.org/resources/papers-and-slides].

[HP13] Florian Holzschuher and René Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In Giovanna Guerrini, editor, *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, pages 195–204. ACM, 2013.

[HP16] Florian Holzschuher and René Peinl. Querying a graph database - language selection and performance considerations. *Journal of Computer and System Sciences*, 82(1):45–68, 2016.

[HS13] Steve Harris and Andy Seaborne. SPARQL 1.1 Overview. https://www.w3.org/TR/sparql11-overview/, March 2013. [Online; last accessed May 2020].

[Kun87] H. S. Kunii. DBMS with graph data model for knowledge handling. In Stephen A. Szygenda, editor, *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 138–142. ACM, 1987.

[LDB20] LDBC Social Network Benchmark task force. The ldbc social network benchmark (version 0.4.0-snapshot). used snapshot version: https://github.com/martin-kl/Diploma_Thesis_01526110_code/blob/master/ldbc-snb-specification_0.4.0-SNAPSHOT_from_2020-11-03.pdf, current LDBC snapshot: https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf, 2020. [Online; last accessed November 2020; used snapshot from Nov. 03 2020].

[Lin18] Tobias Lindaaker. An overview of the recent history of graph query languages. https://s3.amazonaws.com/artifacts.opencypher.org/website/materials/DM32.2/DM32.2-2018-00085R1-recent_

history_of_property_graph_query_languages.pdf, May 2018. [Online; last accessed October 2020; linked from http://www.opencypher.org/references].

[LM13] Katja Losemann and Wim Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems*, 38(4):24:1–24:39, 2013.

[LMV16] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

[LS06] Yanhong A. Liu and Scott D. Stoller. Querying complex graphs. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2006.

[LV12] Leonid Libkin and Domagoj Vrgoc. Regular path queries on graphs with data. In Alin Deutsch, editor, *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 74–85. ACM, 2012.

[MHM17] Jan Michels, Keith Hare, and Jim Melton. Standardizing graph database functionality. https://drive.google.com/file/d/18vlmMAZBFAvBkKipSRiu94JTLCt4vxC8/view, February 2017. [Online; last accessed May 2020].

[MSV17] József Marton, Gábor Szárnyas, and Dániel Varró. Formalising opencypher graph queries in relational algebra. In Marite Kirikova, Kjetil Nørvåg, and George A. Papadopoulos, editors, *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, volume 10509 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2017.

[MW89] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, pages 185–193. Morgan Kaufmann, 1989.

[Neo20] Neo4j, Inc. The Neo4j Cypher manual. https://neo4j.com/docs/cypher-manual/current/, 2020. [Online; last accessed October 2020].

[ope18] openCypher Implementers Group. Cypher query language reference, version 9. https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf, 2018. [Online; last accessed October 2020].

[Ora20] Oracle. PGQL 1.3 specification | property graph query language. https://pgql-lang.org/spec/1.3/, March 2020. [Online; last accessed October 2020].

[PAG10] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. nsparql: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010.

[Pla18] Stefan Plantikow. Summary chart of Cypher, PGQL, and G-Core. https://gql.today/wp-content/uploads/2018/05/ytz-030r1-Summary-Chart-of-Cypher-PGQL-GCore-1.pdf, May 2018. [Online; last accessed November 2020; linked from https://gql.today/comparing-cypher-pgql-and-g-core/].

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF, W3C recommendation. https://www.w3.org/TR/rdf-sparql-query/, January 2008. [Online; last accessed May 2020].

[RABM17] Michael Rath, David Akehurst, Christoph Borowski, and Patrick Mäder. Are graph query languages applicable for requirements traceability analysis? In Eric Knauss, Angelo Susi, David Ameller, Daniel M. Berry, Fabiano Dalpiaz, Maya Daneva, Marian Daun, Oscar Dieste, Peter Forbrig, Eduard C. Groen, Andrea Herrmann, Jennifer Horkoff, Fitsum Meshesha Kifetew, Marite Kirikova, Alessia Knauss, Patrick Maeder, Fabio Massacci, Cristina Palomares, Jolita Ralyté, Ahmed Seffah, Alberto Siena, and Bastian Tenbergen, editors, *Joint Proceedings of REFSQ-2017 Workshops, Doctoral Symposium, Research Method Track, and Poster Track co-located with the 22nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2017), Essen, Germany, February 27, 2017*, volume 1796 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[RH19] Florin Rusu and Zhiyi Huang. In-depth benchmarking of graph database systems with the linked data benchmark council (LDBC) social network benchmark (SNB). *CoRR*, abs/1907.07405, 2019.

[RN11] Marko A. Rodriguez and Peter Neubauer. The graph traversal pattern. In Sherif Sakr and Eric Pardede, editors, *Graph Data Management: Techniques and Applications*, pages 29–46. IGI Global, 2011.

[Rod15] Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In James Cheney and Thomas Neumann, editors, *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, pages 1–10. ACM, 2015.

[RTH+17] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. Pgx.d/async: A scalable distributed graph pattern matching engine. In Peter A. Boncz

and Josep-Lluís Larriba-Pey, editors, *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, pages 7:1–7:6. ACM, 2017.

[RWE15] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2nd edition, 2015.

[Sco06] Michael L. Scott. *Programming language pragmatics (2. ed.)*. Morgan Kaufmann, 2006.

[SHvR+16] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jay Banerjee, and Hassan Chafi. Using domain-specific languages for analytic graph databases. *Proceedings of the VLDB Endowment*, 9(13):1257–1268, 2016.

[SS19] Chandan Sharma and Roopak Sinha. A schema-first formalism for labeled property graph databases: Enabling structured data loading and analytics. In Kenneth Johnson, Josef Spillner, Xinghui Zhao, Olga Datskova, and Blesson Varghese, editors, *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT 2019, Auckland, New Zealand, December 2-5, 2019*, pages 71–80. ACM, 2019.

[SW14] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proc. VLDB Endow.*, 7(7):577–588, 2014.

[The02] Dimitri Theodoratos. Semantic integration and querying of heterogeneous data sources using a hypergraph data model. In Barry Eaglestone, Siobhán North, and Alexandra Poulovassilis, editors, *Advances in Databases, 19th British National Conference on Databases, BNCOD 19, Sheffield, UK, July 17-19, 2002, Proceedings*, volume 2405 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2002.

[Tig] TigerGraph. GSQL language reference. `https://docs.tigergraph.com/v/3.0/dev/gsql-ref`. [Online; last accessed November 2020, GSQL version on TigerGraph 3.0].

[Tin15] TinkerPop3 documentation v3.0.1. `https://tinkerpop.apache.org/docs/3.0.1-incubating/`, September 2015. [Online; last accessed October 2020].

[Tin20] Tinkerpop3 documentation v3.4.6. `http://tinkerpop.apache.org/docs/current/reference/`, February 2020. [Online; last accessed October 2020].

[TKL19] Frank Tetzel, Romans Kasperovics, and Wolfgang Lehner. Graph traversals for regular path queries. In Akhil Arora, Arnab Bhattacharya, and George H. L. Fletcher, editors, *Proceedings of the 2nd Joint International Workshop*

on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Amsterdam, The Netherlands, 30 June 2019, pages 5:1–5:8. ACM, 2019.

[TPAV19] Harsh Thakkar, Dharmen Punjani, Sören Auer, and Maria-Esther Vidal. Towards an integrated graph algebra for graph pattern matching with gremlin (extended version). *CoRR*, abs/1908.06265, 2019.

[TvR19] Vasileios Trigonakis and Oskar van Rest. Property graph extensions for the SQL standard. `http://wiki.ldbcouncil.org/download/attachments/106233859/ldbc_tuc_2019_sql-pgq.pdf`, July 2019. [Online; last accessed November 2020].

[vR17] Oskar van Rest. PGQL - status update and comparison to LDBC's graph QL proposals. `https://github.com/ldbc/tuc_presentations/blob/master/20170209-pgql.pdf`, alternative link: `http://wiki.ldbcouncil.org/download/attachments/59277315/ninth_ldbc_tuc_pgql.pdf`, February 2017. [Online; last accessed November 2020].

[vRHK+16] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In Peter A. Boncz and Josep-Lluís Larriba-Pey, editors, *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES), Redwood Shores, CA, USA, June 24 - 24, 2016*, page 7. ACM, 2016.

[WD18] Mingxi Wu and Alin Deutsch. GSQL: An sql-inspired graphquery language. `https://info.tigergraph.com/gsql`, 2018. [Online; last accessed May 2020].

[Woo90] Peter T. Wood. Factoring augmented regular chain programs. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 255–263. Morgan Kaufmann, 1990.

[Woo12] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[WS90] Carolyn R. Watters and Michael A. Shepherd. A transient hypergraph-based model for data access. *ACM Trans. Inf. Syst.*, 8(2):77–102, 1990.

[Wu18] Mingxi Wu. Property graph type system and data definition language. *CoRR*, abs/1810.08755, 2018.

[Yan90] Mihalis Yannakakis. Graph-theoretic methods in database theory. In Daniel J. Rosenkrantz and Yehoshua Sagiv, editors, *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,*

April 2-4, 1990, Nashville, Tennessee, USA, pages 230–242. ACM Press, 1990.