A Semantics of GQL; a New Query Language for Property Graphs Formalized

O. H. Morra

2021-07-29

Nowadays, graph databases are becoming more prevalent in the industry. Companies such as Neo4j and Amazon Neptune thrive and provide graph database solutions for large companies such as Airbnb and Ebay. However, there is no consensus yet on a query language for graph databases, in contrast to relational databases and its query language SQL.

Past years the fundament has been laid out for a new standard query language by members of the graph database community. Green's manifesto [16] pushed the graph database community to combine three leading property graph query languages and SQL together to obtain a single standard query language for graph databases. Today, this new query language called GQL is being standardized by Plantikow [21].

In this document I will define a formal semantics for GQL and provide an interpreter for simple GQL queries based on this semantics. At this point in time GQL is not yet a fully standardised language by ISO, hence, this semantics is based on the GQL ISO Standard of 02-02-2020 [21] and updated during the term of my project. Additionally, GQL is based on PGQL, Cypher, and G-CORE, and, for the latter two languages there are already semantics defined, respectively by Francis et al. [14] and Angles et al. [10]. Both semantics will provide a basis for the semantics defined for GQL in this document.

The latter graph query language G-CORE is defined by industry and academia in order "to shape the future of graph query languages" [10], however, it is not widely used, i.e. I found only one interpreter [12] in the academic field. On the other hand, Cypher exists for almost a decade and is implemented by Neo4j, so one of the most used graph query languages [22]. In contrast to the defined semantics of Cypher and G-CORE, there is no working implementation of GQL at this point in time. As GQL will most likely be adopted by most vendors as it is an ISO Standard, implementations will arise and probably will differ lightly between implementations similar to SQL implementations [17]. Guagliardo et al. have defined semantics for a small part of SQL and emphasized the need for the semantics being coherent with the available implementations of SQL [17]. Also, Francis et al. have adopted this view [13]. Therefore, I will adopt a similar strategy on defining a new semantics, i.e. I will stay close to the GQL

ISO Standard [21] and ensure the formal definitions are generally applicable and flexible for multiple implementation dependent features.

Additionally, it is not possible to fully formalize the GQL Standard to a semantics during my internship. Therefore, the semantics will encompass reading statements from GQL. More precisely, based on the data model proposed in Sec. 1 `FROM-MATCH-RETURN-WHERE` queries in the most basic form will be formalized. Besides, only rigid graph patterns, patterns with a fixed path length, will be considered in the `MATCH` clause. These rigid graph patterns can be mandatory or optional, and have four different evaluation configurations which will be discussed. Variable length graph patterns would elaborate the formal semantics significantly and require a discussion on its own.

Not only will the language be statically formalized, also, a naive implementation, based on the formal semantics, of a GQL query interpreter will be provided. Guagliardo et al. [17] provide a similar interpreter for SQL in order to validate their proposed semantics against current implementations of SQL. The aim of the GQL interpreter is slightly different. The aim is to demonstrate the syntax and semantics of GQL and contribute an interactive tool for academics, developers and other parties of interest as there is no working implementation of GQL. The naive implementation will present query results as well as yield the proposed formalization of the query. This way queries can be compared on an algebraic level instead of the syntactic level. One can find the implementation on GitHub [5].

Accordingly, my report will be structured as follows. In Section 1 I will expand on the underlying data model defined for GQL, describing formal definitions for values, property graphs, a graph database and binding tables. Section 2 will focus on pattern matching by giving the first part of the GQL syntax, explaining how the syntax relates to patterns and how patterns are matched on a property graph. Subsequently, in Section 3 the remainder of the syntax will be given, that is the syntax of expressions, queries and clauses. Following the complete syntax, in Section 4 the semantics of each part of the syntax will be proposed. In section 5 I will expand on the naive implementation of the GQL interpreter. Section 6 will provide an overview of the contributions of my research and propose several new directions for future research. At last, in Section 7 I will present an conclusive overview of my research.

1 Data Model

A query language cannot exist without a data model to execute its queries on. In this chapter I propose the data model described in the GQL ISO Standard [21]. The data model consists of values, described in Section 1.1, property graphs, described in Section 1.2, a graph database, described in Section 1.3, and binding tables, described in Section 1.4. Joining all the previous definitions, the GQL data model is obtained.

1.1 Values

In a GQL environment there are multiple sets, namely $\mathcal{N}$ a set of node identifiers, $\mathcal{E}$ a set of edge identifiers, $\mathcal{K}$ a set of property keys, $\mathcal{A}$ a set of names, which are all countably infinite, and $D = \{0, 1\}$ defining direction of an edge ($0 =$ undirected, $1 =$ directed). For this representation of the model we assume three base types: the type of real numbers $\mathbb{R}$, the type of finite strings over a finite alphabet Σ and the type of floating point numbers $\mathbb{F}$.

As the scope of this semantics is to define read statements in **FROM-MATCH-RETURN-WHERE** clauses, it is not focused on comparison between specific data types. However, it is of importance how such comparison is done, hence, in Section 4.1.4 I elaborate on data type comparison. Now, the set $\mathcal{V}$ of values that is needed for this semantics is a subset of all values defined by the GQL ISO Standard [21]. So, the set $\mathcal{V}$ of values is defined as follows:

- Identifiers (i.e., elements of $\mathcal{N}$ and $\mathcal{E}$) are values;
- Base types (elements of $\mathbb{R}$, $\mathbb{F}$ and Σ^*) are values;
- **true**, **false**, **null** and **unknown** are values where **null** and **unknown** are equivalent;
- $\mathsf{set}()$ is a value (empty set), and if $v_1, \ldots, v_m$ are distinct values (i.e., is duplicate-free), for $m > 0$, then $\mathsf{set}(v_1, \ldots, v_m)$ is a value;
- $\mathsf{multiset}()$ is a value (empty multiset), and if $v_1, \ldots, v_m$ are values, for $m > 0$, then $\mathsf{multiset}(v_1, \ldots, v_m)$ is a value;
- $\mathsf{map}()$ is a value (empty map), and if $k_1, \ldots, k_m$ are distinct property keys and $v_1, \ldots, v_m$ are values, for $m > 0$, then $\mathsf{map}((k_1, v_1), \ldots, (k_m, v_m))$ is a value;
- if n is a node identifier, then $\mathsf{path(n)}$ is a value. If $n_1, \ldots, n_m$ are node identifiers and $e_1, \ldots, e_{m-1}$ are edge identifiers, for $m > 1$ and for each $e_i \in \{e_1, \ldots, e_{m-1}\}$ the source node is n_i and the target node is n_{i+1}, then $\mathsf{path}(n_1, e_1, n_2, \ldots, n_{m-1}, e_{m_1}, n_m)$ is a value. From here paths will be abbreviated to $n_1 e_1 n_2 \cdots n_{m-1} e_{m-1} n_m$.

In the syntax of GQL, sets are denoted as $S = \{v_1, \ldots, v_m\}$, multisets are denoted as $(S, i) = \{v_1^{i(v_1)}, \ldots, v_m^{i(v_m)}\}$ where upper indices indicate the multiplicity of a value and when equal to 1 are omitted, and maps are denoted as $\{k_1 : v_1, \ldots, k_m : v_m\}$. This explicit notation will be used to distinguish the syntax from the semantics of values. Note that regular and distinct lists are left out because there equality is not yet properly defined and order on tables is not defined within the scope of this internship.

Similarly to Francis et al. [13] the symbol "$\cdot$" will denote conatenation of paths, which is only possible if the first path ends in node n and the second path starts from node n, i.e. if $p_1 = n_1 e_1 \cdots e_{j-1} n_j$ and $p_2 = n_j e_j \cdots e_{m-1} n_m$ then $p_1 \cdot p_2 = n_1 e_1 \cdots e_{j-1} n_j e_j \cdots e_{m-1} n_m$.

Additionaly, Francis et al. [13] define "a finite set F of predefined functions that can be applied to values (and produce new values)", like concatenation of strings and arithmetics on numbers. They do so in order to parameterize the semantics using F, such that the semantics "can be extended whenever new types and/or basic functions are added to the language". This is in line with the approach of this semantics for GQL and is useful when GQL will eventually be used more widely, hence, such a set F of predefined functions is also defined in the semantics for GQL.

1.2 Property graphs

Informally, a property graph is a directed labelled multigraph with the special characteristic that each node or edge could maintain a set (possibly empty) of property-value pairs. [9]

Although there was no formal definition for property graphs, there were multiple querying languages available for property graphs (e.g. Cypher, G-Core). Angles [9] proposed a simple, but formal definition of the property graph database model in 2018. In the book Querying Graphs by Bonifati et al. another version of the property graph database model is given with multiple extensions [11], also in 2018. It is clear that there are multiple implementations possible, and looking into the GQL ISO Standard [21], there are different options which I will formally disclose below. I adapt the formal definition from Angles [9] to match the different implementations described in the GQL ISO Standard [21].

Let $\mathcal{L}$ be a countable set of labels, $\mathcal{P}$ be a countable set of key-value pairs of the form (k, v) where $k \in \mathcal{K}$ and $v \in \mathcal{V}$. For a given set $\mathcal{X}$ I assume that $\text{SET}^+(\mathcal{X})$ is the set of all finite subsets of $\mathcal{X}$, without the empty set. Also, I assume that $\text{SET}^1(\mathcal{X})$ is the set of all singleton subsets of $\mathcal{X}$, without the empty set. Then the following definition is constructed:

Definition 1.1 (Property graph [9]) *A property graph is a tuple $G = (N, E, \eta, \varepsilon, \lambda_N, \lambda_E, \phi)$ with a name $a \in \mathcal{A}$ where:*

1. *N is a finite subset of node identifiers $\mathcal{N}$ (also called vertices);*
2. *E is a finite subset of edge identifiers $\mathcal{E}$;*
3. *$\eta : E \rightarrow (N \times N)$ is a total function that associates each edge in E with a pair of nodes in N where the first node is its source and the second node its target (i.e., η is the usual incidence function in graph theory). The source and target node may be the same node;*
4. *$\varepsilon : E \rightarrow D$ is a surjective function that defines each edge as undirected or directed;*
5. *λ_N is a labeling function for nodes where it is implementation dependent which function(s) is/are valid:*

   (a) *$\lambda_N : N \to \emptyset$ is a partial function that associates a node with an empty set*
   (b) *$\lambda_N : N \to \mathrm{SET}^1(\mathcal{L})$ is a partial function that associates a node with a singleton label set from $\mathcal{L}$*
   (c) *$\lambda_N : N \to \mathrm{SET}^+(\mathcal{L})$ is a partial function that associates a node with a set of one or more labels from $\mathcal{L}$*

   A combination of the above functions can be implemented in parallel, e.g. a partial function that associates a node with a set of zero or more labels is defined as $\lambda_N : N \to \mathrm{SET}^+(\mathcal{L}) \cup \{\emptyset\}$;

6. *λ_E is a labeling function for edges where it is implementation dependent which function(s) is/are valid:*
   (a) *$\lambda_E : E \to \emptyset$ is a partial function that associates an edge with an empty set*
   (b) *$\lambda_E : E \to \mathrm{SET}^1(\mathcal{L})$ is a partial function that associates an edge with a singleton label set from $\mathcal{L}$*
   (c) *$\lambda_E : E \to \mathrm{SET}^+(\mathcal{L})$ is a partial function that associates an edge with a set of one or more labels from $\mathcal{L}$*

   A combination of the above functions can be implemented in parallel, e.g. a partial function that associates an edge with a set of zero or more labels is defined as $\lambda_E : E \to \mathrm{SET}^+(\mathcal{L}) \cup \{\emptyset\}$;

7. *$\phi : (N \cup E) \to \mathrm{SET}^+(\mathcal{P}) \cup \{\emptyset\}$ is a partial function that associates nodes/edges with a set of property key-value pairs from $\mathcal{P} \cup \{\emptyset\}$.*

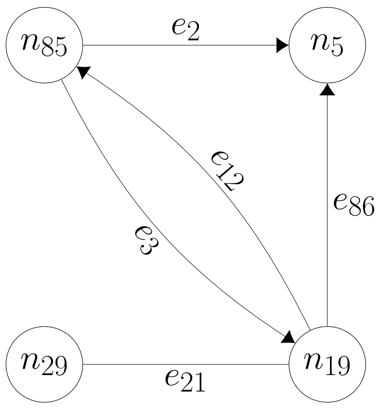
An implementation should by default support directed edges. Also, it should ensure that each label or property is unique within a graph, node or edge. Besides, the maximum cardinality of labels and properties is implementation defined.

1.2.1 Example property graphs

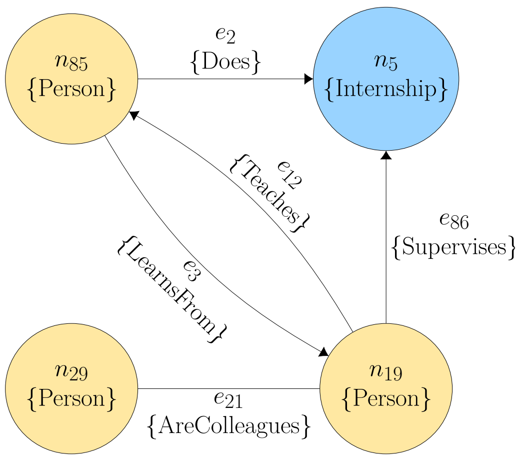
Clearly, the definition of a property graph in GQL leaves multiple details up to the implementation. It is obvious that property graphs can have different forms across implementations following the definition. However, various implementations can lead to incompatible queries across those implementations, hence, understanding their differences is essential. In order to clarify what impact such implementation differences can have, three example property graphs are shown in Fig. 1.

All three property graphs in Fig. 1 are constructed from the following sets and relations following Def. 1.1:

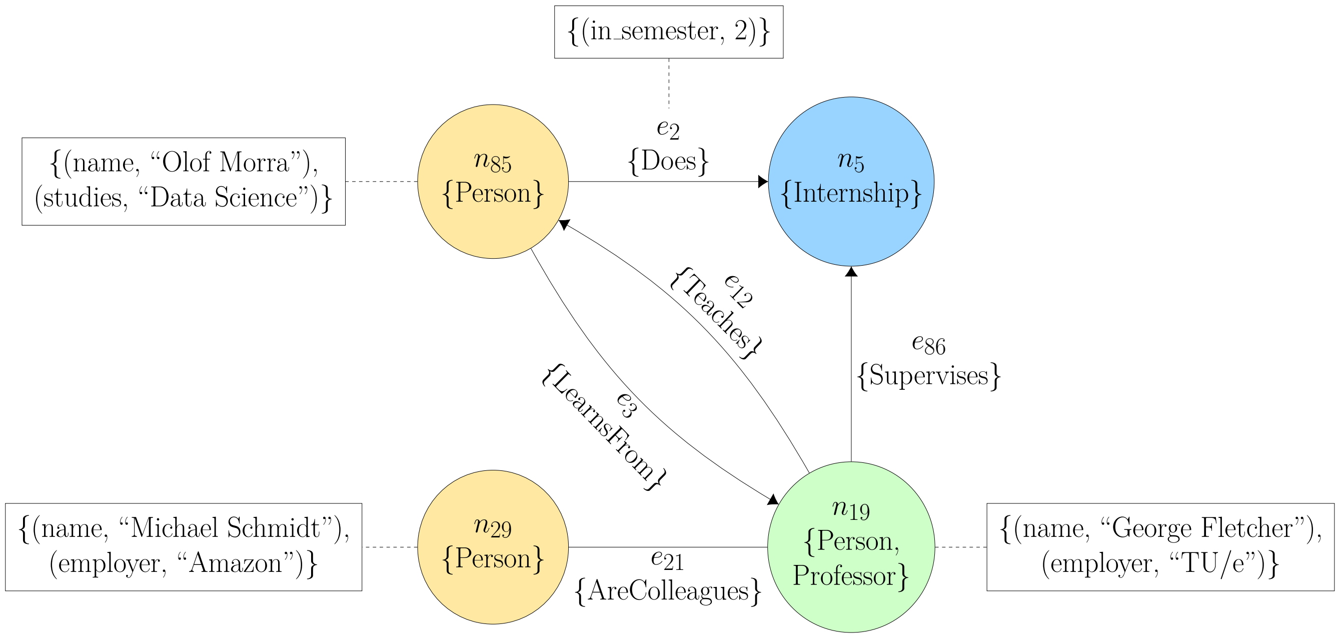
1. $N = \{n_5, n_{19}, n_{29}, n_{85}\}$;



(a) Basic property graph G_1



(b) Property graph G_2 with singleton label sets



{(in_semester, 2)}
{(name, “Olof Morra”), (studies, “Data Science”)}
n_{85} {Person}
e_2 {Does}
n_5 {Internship}
e_{12} {Teaches}
e_{86} {Supervises}
e_3 {LearnsFrom}
{(name, “Michael Schmidt”), (employer, “Amazon”)}
n_{29} {Person}
e_{21} {AreColleagues}
n_{19} {Person, Professor}
{(name, “George Fletcher”), (employer, “TU/e”)}

(c) Property graph G_3 with unbounded label sets and property sets

Figure 1: Three property graphs

2. $E = \{e_2, e_3, e_{12}, e_{21}, e_{86}\}$;
3. $\mathcal{L}$ = {Person, Internship, Professor, Does, Teaches, LearnsFrom, Supervises, AreColleagues};
4. P = {(name, "Olof Morra"), (name, "George Fletcher"), (name, "Michael Schmidt"), (studies, "Data Science"), (employer, "TU/e"), (employer, "Amazon"), (in_semester, 2)}
5. $\eta : E \to (N \times N)$ is defined by $\eta(e_2) = (n_{85}, n_5)$, $\eta(e_3) = (n_{85}, n_{19})$, $\eta(e_{12}) = (n_{19}, n_{85})$, $\eta(e_{21}) = (n_{29}, n_{19})$ and $\eta(e_{86}) = (n_{19}, n_5)$;
6. $\varepsilon : E \to D$ is defined by $\varepsilon(e_2) = \varepsilon(e_3) = \varepsilon(e_{12}) = \varepsilon(e_{86}) = 1$ and $\varepsilon(e_{21}) = 0$.

However, they do not necessarily have to be supported by each implementation of GQL, so I will demonstrate the differences across implementations via property graphs G_1, G_2, G_3, which are their respective names. The node label function λ_N for property graph G_1 is defined as $\lambda_N : N \to \emptyset$ as each node only has an identifier. For property graph G_2 $\lambda_N : N \to \text{SET}^1(\mathcal{L})$ is defined by $\lambda_N(n_{19}) = \lambda_N(n_{29}) = \lambda_N(n_{85}) = \{Person\}$ and $\lambda_N(n_5) = \{Internship\}$. At last, for property graph G_3 $\lambda_N : N \to \text{SET}^+(\mathcal{L})$ is defined by $\lambda_N(n_{29}) = \lambda_N(n_{85}) = \{Person\}$, $\lambda_N(n_{19}) = \{Person, Professor\}$ and $\lambda_N(n_5) = \{Internship\}$.

Similarly, for property graph G_1 the edge label function is defined as $\lambda_E : E \to \emptyset$ as each edge only has an identifier. For property graph G_2 and G_3 we have $\lambda_E : E \to \text{SET}^1(\mathcal{L})$ defined as $\lambda_E(e_2) = \{Does\}$, $\lambda_E(e_3) = \{LearnsFrom\}$, $\lambda_E(e_{12}) = \{Teaches\}$, $\lambda_E(e_{21}) = \{AreColleagues\}$ and $\lambda_E(e_{86}) = \{Supervises\}$.

Furthermore, property graph G_1 and G_2 do not contain any properties, so we define $\phi : (N \cup E) \to \text{SET}^+(\mathcal{P}) \cup \emptyset$ as $\phi(ne) = \emptyset$ for all $ne \in N \cup E$ for both property graphs. On the other hand, property graph G_3 does contain properties. So $\phi : (N \cup E) \to \text{SET}^+(\mathcal{P}) \cup \emptyset$ for property graph G_3 is defined as $\phi(2)$ = {(in_semester, 2)}, $\phi(n_{19})$ = {(name, "George Fletcher"), (employer, "TU/e")}, $\phi(n_{29})$ = {(name, "Michael Schmidt"), (employer, "Amazon")}, $\phi(n_{85})$ = {(name, "Olof Morra"), (studies, "Data Science")} and $\phi(ne) = \emptyset$ for all other $ne \in N \cup E$.

Looking at the definitions of each graph, it is clear that G_1 contains a similar structure as G_2 each node or edge in G_1 is also present in G_2 without changing its structure, similarly G_2 has a similar structure as G_3. However, the other way around this does not hold, clearly node and edge labels cannot be defined in G_1 and there is no possibility of adding multiple labels to nodes or edges in G_2. As each graph in Fig. 1 can be formed by a different implementation, such lack of similar structure can lead to incompatible queries between implementations. Below a fraction of a query containing a match clause is given to exemplify such an incompatibility:

```
MATCH (a:Person)-[:Supervises]->(b:Person)
```

This query wants to match a pattern containing two nodes, both labeled as *Person*, and share a directed relation named *Supervises*. This pattern exists in G_2 and G_3, for node pair (n_{19}, n_5) the pattern is satisfied in both graphs, however, in property graph G_1 the node and edge label functions both have $\emptyset$ as range. Hence, the query is invalid on graph G_1 as the query looks for labels but G_1 does not support labels on nodes and edges. Hence, implementations of GQL should explicitly define their implementation of property graphs.

1.3 Graph database

Now, GQL does allow the user to interact with multiple graphs, a feature not discussed by Francis et al. [13], but also not as elaborate as interacting with multiple tables in a SQL database defined by Guagliardo et al. [17]. For the scope of this internship it suffices to have a schema $S \subset \mathcal{A}$ of graph names and to define a graph database D that it maps each name $s \in S$ to a graph G.

1.4 Binding tables

Although GQL is a query language for property graphs, it also supports binding tables as output and uses such tables for intermediary results. Such a table is similar to an SQL table and Cypher table, but also has some key differences that give a binding table more flexibility in form, but rigidity in the use after construction than the SQL table. Most notably, a binding table is immutable after construction.

Then, the definition of a record is similar to Francis et al. [13]. Hence, a record is a partial function from names to values denoted as a tuple with named fields $r = (a_1 : v_1, \ldots, a_n : v_n)$ where $a_1, \ldots, a_n \in \mathcal{A}^2$ are distinct names, and $v_1, \ldots, v_n \in \mathcal{V}$ are values. Similar to Guagliardo et al. [17] I define each name a_i with $i \in \{1, \ldots, n\}$ from the set of pairs of names $\mathcal{A}^2$, i.e. a full name, which is written as $a_{i1}.a_{i2}$ instead of (a_{i1}, a_{i2}) where all a_{i1} refer to a graph name in schema S and a_{i2} refers to a node or edge pattern name. I refer to $\mathsf{dom}(r)$, i.e. the domain of r, as the set $\{a_1, \ldots, a_n\}$ of names used in r. The unit value is a record $r = ()$ with zero fields. For brevity, if a query is executed on one graph $D(s)$, $s.a_i$ is abbreviated to a_i. In contrast to the formal definition of binding tables in Cypher by Francis et al. [13], the order in which the fields appear can be defined by σ. Additionaly, we have function $\delta : T \to T$ that maps a table to itself with all duplicates eliminated. The following definition is constructed:

Definition 1.2 (Binding table) *A binding table is a tuple* $T = (A_T, R, \sigma)$ *where:*

1. A_T *is a finite subset of full names* $\mathcal{A}^2$*;*
2. R *is a finite subset of records where for each* $r \in R$*,* $\mathsf{dom}(r) = A_t$*;*
3. $\sigma : \{A_T\} \to \{1, \ldots, |A_T|\}$ *is a sorting function on* A_T*.*

An implementation should by default declare a binding table that allows duplicates and without a sorting function σ. The sorting function σ is optional, if it does not exist the table does not have a preferred column sequence. Following this definition I can obtain two values that can store a binding table, which indicates that combining different types of binding tables needs to be clarified.

1.4.1 Example binding table

Now, let us have the following three match queries on graph G_3 (see Fig. 1(c)):

1. `MATCH (p1:Person)-[:Teaches]->(p2:Person)`
2. `MATCH (p2:Person)-[:LearnsFrom]->(p1:Person)`
3. `MATCH (p1:Person)-[:AreColleagues]-(p2:Person)`

Then I obtain the two following binding tables without sorting function σ and only writing a_{i2} for each name as the match queries are on the same graph (so a_{i1} is implicit and equal to G_3):

1. $T_1 = (\{p1, p2\}, \{(p1 : n_{19}, p2 : n_{85})\})$
2. $T_2 = \delta((\{p1, p2\}, \{(p1 : n_{19}, p2 : n_{85})\}))$
3. $T_3 = (\{p1, p2\}, \{(p1 : n_{19}, p2 : n_{29})\})$

Now, I will union the three tables T_1, T_2 and T_3 in two different ways and show that the outcome T_4 is different depending on the definition of a binding table. More specifically, I obtain two different tables T_4 which are all a valid GQL union of T_1, T_2 and T_3, namely:

1. As set: $T_4 = \delta((\{p1, p2\}, \{(p1 : n_{19}, p2 : n_{85}), (p1 : n_{19}, p2 : n_{85}), (p1 : n_{29}, p2 : n_{19})\}))$
2. As multiset: $T_4 = (\{p1, p2\}, \{(p1 : n_{19}, p2 : n_{85}), (p1 : n_{19}, p2 : n_{85}), (p1 : n_{29}, p2 : n_{19})\})$

From these definitions I obtain the following multisets:

1. $\{(p1 : n_{19}, p2 : n_{85}), (p1 : n_{29}, p2 : n_{19})\}$
2. $\{(p1 : n_{19}, p2 : n_{85})^2, (p1 : n_{29}, p2 : n_{19})\}$

Now in both cases I did not define a sorting function σ, or implicitly $\sigma = \{\{p1, p2\} \mapsto \{1, 2\}\}$. Clearly, the mapping can be changed in one way, namely, $\sigma = \{\{p1, p2\} \mapsto \{2, 1\}\}$ by interchanging the two columns.

2 Pattern matching

In this section I will discuss the syntax of patterns, satisfaction of rigid patterns, pattern matching of rigid patterns and matching tuples of rigid path patterns. This means that I do not discuss variable length patterns as that is out of scope for this project. Similar to the Cypher grammar the GQL grammar is defined by mutual recursion of expressions, patterns, clauses and queries.

2.1 Syntax of patterns

In Fig. 2 the GQL syntax can be found in a concise and safe form, i.e. only allowing syntactically correct queries. For example, a choice between two or three types of edge patterns is syntactically defined in the GQL ISO Standard. However, it does not add any expressiveness to GQL as a union of different type of edges with the same element pattern filler has the same expressive power. Additionally, all edge patterns have an abbreviated form for empty, so without element pattern filler, edges which are not included in the given syntax.

Not only are abbreviations left out, additionally, the syntax is limited to the desired scope. Some parts that are left out are element pattern cost clauses, graph pattern yield and keep clauses, path search prefixes, **PATH** or **PATHS** in path pattern prefix, path multiset alternation, path union and questioned path primary. All these elements do attribute other capabilities to GQL, but do not fit within the scope.

$$
\begin{array}{rcll}
\mathsf{path\_pattern} & ::= & (p\ \texttt{=})?\ \mathsf{path\_pattern\_prefix}?\ \mathsf{path\_pattern\_expr} & p \in \mathcal{A} \\
\mathsf{path\_pattern\_prefix} & ::= & \texttt{WALK} \mid \texttt{TRAIL} \mid \texttt{ACYCLIC} \mid \texttt{SIMPLE} & \\
\mathsf{path\_pattern\_expr} & ::= & \mathsf{node\_pattern}\ (\ \mathsf{edge\_pattern}\ \mathsf{node\_pattern})* & \\
\mathsf{node\_pattern} & ::= & \texttt{(}\ \mathsf{elt\_pattern\_filler}\ \texttt{)} & \\
\mathsf{edge\_pattern} & ::= & (\texttt{<-[}\ \mathsf{elt\_pattern\_filler}\ \texttt{]-} \mid \texttt{~[}\ \mathsf{elt\_pattern\_filler}\ \texttt{]~} & \\
 & & \mid \texttt{-[}\ \mathsf{elt\_pattern\_filler}\ \texttt{]->})\ \mathsf{len}? & \\
\mathsf{elt\_pattern\_filler} & ::= & a?\ \mathsf{is\_label\_expr}?\ \mathsf{elt\_predicate}? \qquad a \in \mathcal{A} & \\
\mathsf{is\_label\_expr} & ::= & (\texttt{IS} \mid \texttt{:})\ \mathsf{label\_expr} & \\
\mathsf{elt\_predicate} & ::= & \mathsf{where\_clause} \mid \{\ \mathsf{property\_list}\ \ \} & \\
\mathsf{property\_list} & ::= & k\texttt{:}\mathsf{expr}\ (\texttt{,}\ k\texttt{:}\mathsf{expr})* \qquad k \in \mathcal{K} & \\
\mathsf{len} & ::= & \{\ q \mid q_1\ \texttt{,}\ q_2\ \} \qquad q, q_1 \in \mathbb{N}, q_2 \in \mathbb{N}^+, q_1 = q_2 &
\end{array}
$$

Figure 2: GQL pattern syntax

The GQL syntax of patterns is given in Fig. 2. The symbols highlighted in red denote tokens of the GQL language. However, this syntax is not appropriate for a formal definition of GQL, the aim of this report. Hence, an abstract mathematical notation will be given as formal algebra of GQL. Moreover, the syntax of GQL is very similar as that of Cypher and PGQL (as expected, GQL is based on those two languages). Therefore, to remain coherent in the field of graph database languages, multiple abstract mathematical notations will be taken from Francis et al. [13] and will be altered slightly to fit the presented data model.

However, not all syntactic behaviour will be allowed in the presented data model, such as label negation, nested label expressions and where clauses within elements. In Appendix A I give methods to rewrite queries containing such behaviour to queries that can be mapped to the given data representation.

So, let a node pattern χ be a triple (a, L, P), defined by Francis et al. [13], where:

Definition 2.1 (Node pattern) *Let a node pattern χ be a triple (a, L, P) where:*

1. *$a \in \mathcal{A} \cup \{\mathsf{nil}\}$ is an optional name;*

2. *$L \subset \emptyset \cup \{L_1, \ldots, L_n\}$ for which each $L_i \subset \mathcal{L} \cup \{\%\}$ is a possibly empty finite set of node labels with $i \in \{1, \ldots, n\}$ and $\%$ denoting the wildcard label;*

3. *P is a possibly empty finite set of key-value pairs of the form (k, v) where $k \in \mathcal{K}$ and $v \in \mathcal{V}$.*

The second condition allows to have multiple sets of labels per pattern. This way a pattern match can be matched to at least one set of the set of labels, allowing a label union. Let me illustrate the formalization with the following two equivalent node patterns in GQL syntax:

```
(x:Person|Professor {name: expr₁, employer: expr₂})
(x IS Person|Professor {name: expr₁, employer: expr₂})
```

which are both represented as $(x, \{\{Person\}, \{Professor\}\}, \{(name : ex_1), (employer : ex_2)\})$ where ex_1 and ex_2 respectively represent expressions expr_1 and expr_2. The simplest node pattern () is represented by $(\mathsf{nil}, \emptyset, \emptyset)$.

Now I define an edge pattern based on the relationship pattern by Francis et al. [13].

Definition 2.2 (Edge pattern) *Let an edge pattern ρ be a tuple (d, a, L, P, I) where:*

1. *$d \in \{\rightarrow, \leftarrow, -\}$ specifies the direction of the edge: left-to-right ($\rightarrow$), right-to-left ($\leftarrow$), or undirected ($-$);*

2. *$a \in \mathcal{A} \cup \{\mathsf{nil}\}$ is an optional name;*

3. *$L \subset \emptyset \cup \{L_1, \ldots, L_n\}$ for which each $L_i \subset \mathcal{L} \cup \{\%\}$ is a possibly empty finite set of edge labels with $i \in \{1, \ldots, n\}$ and $\%$ denoting the wildcard label;*

4. *P is a possibly empty finite set of key-value pairs of the form (k, v) where $k \in \mathcal{K}$ and $v \in \mathcal{V}$;*

5. *I is (m, n) with $m, n \in \mathbb{N}$ where $m = n$.*

Similar to the label set L in a node pattern, also the label set L in an edge pattern allows for multiple label sets in a single pattern enabling a label union. In Table 1 you can find examples of edge patterns with different properties and their corresponding representation. In each example the optional grammar token len is defined differently to show its relation to I. If the optional grammar token len does not appear in the syntax of the pattern I is equal to $(1, 1)$, as shown in the first pattern in Table 1. Otherwise, I is equal to (q, q) or (q_1, q_2), respectively shown in the second and third edge pattern in Table 1. So I defines the range of the edge pattern or path pattern. The range is $[m, n]$ if $I = (m, n)$ and $[1, 1]$ if $I = \mathsf{nil}$. Notice that this only allows for rigid patterns, i.e. patterns of fixed length, so with $m = n \in \mathbb{N}^+$ for $I = (m, n)$, and not for variable length patterns, i.e. $m < n \in \mathbb{N}^+$ for $I = (m, n)$. Variable length patterns are allowed in GQL, however, introduce much more complexity to the formal definition, so they are not included in this formal algebra due to a time limit.

Pattern	**Representation**
`-[y:Does {in_semester:2}]->`	$(\rightarrow, y, \{\{Does\}\}, \{(in\_semester, 2)\}, (1, 1))$
`<-[:AreColleagues]-{2}`	$(\leftarrow, \mathsf{nil}, \{\{AreColleagues\}\}, \emptyset, (2, 2))$
`~[IS AreColleagues]~{2,2}`	$(-, \mathsf{nil}, \{\{AreColleagues\}\}, \emptyset, (2, 2))$

Table 1: Example of edge patterns and their representations

Using the above two definitions path patterns can be constructed equivalently to the definition by Francis et al. [13]. Hence, a path pattern π is an alternating sequence of the form

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

where each χ_i is a node pattern and each ρ_j is an edge pattern. A path pattern can have a name a, written as π/a, which is referred to as a named pattern. A path pattern is rigid if all edge patterns in it are rigid, which conforms to the scope of this algebra where only rigid patterns are discussed.

In the next section the satisfaction relation for path patterns w.r.t. a property graph $G = (N, E, \eta, \varepsilon, \lambda_N, \lambda_E, \phi)$, a path with node ids from N and edge ids from E, and an assignment u that maps full names $\mathcal{A}^2$ to values will be defined. The satisfaction relation $\vDash$ for path patterns is dependent on the chosen evaluation mode in the graph pattern. There are four different evaluation modes which define the morphism between the graph G and the path pattern π. In Table 2 each mode (set via path_pattern_prefix), corresponding relation and corresponding morphism is listed. So the four possible notations of a path pattern are $\vDash_W, \vDash_T, \vDash_A$ and $\vDash_S$. If no mode is specified, which is allowed in the given grammar, implicitly the mode is set to **WALK**. First, homomorphic pattern matching ($\vDash_W$) will be discussed, then edge-isomorphism ($\vDash_T$) and at last the two slightly different forms of node-isomorphism ($\vDash_A$ and $\vDash_S$).

Evaluation mode	**Relation**	**Morphism**
WALK	$\vDash_W$	Homomorphism
TRAIL	$\vDash_T$	Edge-isomorphism
ACYCLIC	$\vDash_A$	Node-isomorphism
SIMPLE	$\vDash_S$	Partial node-isomorphism

Table 2: Evaluation modes and their corresponding subscripts and morphisms

2.2 Satisfaction of rigid patterns

The satisfaction of rigid patterns will be defined in four sections, each section describes one evaluation mode and gives a formal definition of the satisfaction relation for that mode. In the first section you can find the broadest definition, whilst, in each other section you find a restriction that must be imposed on the first definition.

As the mathematical notations for nodes and edges are very similar to the notation introduced by Francis et al. [13], also, the notation for the satisfaction of rigid patterns will be alike. Hence, the definition for satisfaction of rigid patterns in Section 4.2 from Francis et al. [13] is adjusted slightly to adhere to the data model definitions and different evaluation modes given in this document.

2.2.1 WALK

The first evaluation mode, **WALK**, is the broadest definition in the sense that it matches a pattern to the most paths in a graph G compared to other evaluation modes. In mathematical terms, it defines a homomorphism between the graph G and path pattern π, i.e. both nodes and edges are allowed to occur multiple times in a path that matches a pattern. Still, a similar inductive definition is obtained for the satisfaction relation $\vDash_W$ as in the paper of Francis et al. [13]. The base case is given by node patterns. Let χ be a node pattern (a, L, P) and path p be a node n, then $(p, G, u) = (n, G, u) \vDash_W \chi$ if all of the following hold:

- either a is nil or $u(a) = n$;
- $L = \emptyset$ or there exists at least one $L_i \in L$ for which $L_i \subseteq \lambda_N(n)$ or $\% \in L_i$ and $\lambda_N(n) \neq \emptyset$;
- $P \subseteq \phi(n)$.

Example 2.1 *Consider the property graph G_3 in Figure 1(c) and the node patterns $\chi_1 = (x, \{\{Person\}\}, \{(name, \text{“Olof Morra”})\})$ and $\chi_2 = (y, \emptyset, \emptyset)$.*

Then, for χ_1 we have:

$$\begin{aligned}
(n_5, G_3, u) &\not\models_W \chi_1 && \textit{for any assignment } u,\\
(n_{19}, G_3, u) &\not\models_W \chi_1 && \textit{for any assignment } u,\\
(n_{29}, G_3, u) &\not\models_W \chi_1 && \textit{for any assignment } u,\\
(n_{85}, G_3, u) &\models_W \chi_1 && \textit{if } u \textit{ is an assignment that maps } x \textit{ to } n_{85}.
\end{aligned}$$

For χ_2 we have

$$\begin{aligned}
(n_5, G_3, u) &\models_W \chi_2 && \textit{if } u \textit{ is an assignment that maps } y \textit{ to } n_5,\\
(n_{19}, G_3, u) &\models_W \chi_2 && \textit{if } u \textit{ is an assignment that maps } y \textit{ to } n_{19},\\
(n_{29}, G_3, u) &\models_W \chi_2 && \textit{if } u \textit{ is an assignment that maps } y \textit{ to } n_{29},\\
(n_{85}, G_3, u) &\models_W \chi_2 && \textit{if } u \textit{ is an assignment that maps } y \textit{ to } n_{85}.
\end{aligned}$$

For the inductive case, let χ be a node pattern, let p be a path with starting node n_p, let π be a rigid path pattern, and let ρ be the edge pattern (d, a, L, P, I). Let I be defined by (m, m) with $m \in \mathbb{N}$. For $m = 0$, we have that $(n \cdot p, G, u) \models_W \chi\rho\pi$ if

- either a is nil or $u(a) = \mathsf{set}()$,
- $(n, G, u) \models_W \chi$,
- $(p, G, u) \models_W \pi$,
- and $n = n_p$.

For $m \geq 1$, we have that $(n_1 e_1 \cdots e_m n_{m+1} \cdot p, G, u) \models_W \chi\rho\pi$ if all of the following hold:

- either a is nil or $u(a) = \mathsf{set}(e_1, \ldots, e_m)$,
- $(n_1, G, u) \models_W \chi$,
- $(p, G, u) \models_W \pi$,
- and $n_{m+1} = n_p$;

and for every $i \in \{1, \ldots, m\}$, all of the following hold:

- $L = \emptyset$ or there exists at least one $L_j \in L$ for which $L_j \subseteq \lambda_E(e_i)$ or $\% \in L_j$ and $\lambda_E(e_i) \neq \emptyset$,
- $P \subseteq \phi(e_i)$,
- $\eta(e_i) \in \begin{cases} \{(n_i, n_{i+1})\} & \text{if } d \text{ is } \rightarrow \\ \{(n_{i+1}, n_i)\} & \text{if } d \text{ is } \leftarrow, \\ \{(n_i, n_{i+1}), (n_{i+1}, n_i)\} & \text{if } d \text{ is } - \end{cases}$

- $\varepsilon(e_i) = \begin{cases} 1 & \text{if } d \text{ is } \rightarrow \text{ or } \leftarrow \\ 0 & \text{if } d \text{ is } - \end{cases}$.

Example 2.2 *Consider the property graph G_3 in Figure 1(c) and the following rigid pattern π in GQL syntax:*

```
(x:Person {name: "Olof Morra"}) -[:Does]-> (y)
```

In the defined mathematical representation the rigid pattern π is presented as:

$$\underbrace{(x, \{\{Person\}\}, \{(name, \text{“}Olof Morra\text{”})\})}_{\chi_1}\underbrace{(\rightarrow, \mathit{nil}, \{\{Does\}\}, \emptyset, (1,1))}_{\rho}\underbrace{(y, \emptyset, \emptyset)}_{\chi_2}$$

where χ_1 and χ_2 are the node patterns we have seen in Example 2.1. Now, let $u = \{x \mapsto n_{85}, y \mapsto n_5\}$, so we have that $(n_{85}, G_3, u) \vDash_W \chi_1$ and $(n_5, G_3, u) \vDash_W \chi_2$. Using the above satisfaction definition we obtain that $(p, G, u) \vDash_W \pi$, where $p = n_{85}e_2n_5$ and $\pi = \chi_1\rho\chi_2$.

2.2.2 TRAIL

The second satisfaction relation for path patterns is $\vDash_T$, defining an edge-isomorphism between the graph G and path pattern π. This definition is more restricted then the previous relation as now edges must be distinct in a path p. So, $(p, G, u) \vDash_T \pi$ for a path p and path pattern π is equal to the definition of $\vDash_W$ with the precondition that all edges in p are distinct. This evaluation mode is the same as in Cypher.

2.2.3 ACYCLIC

The third satisfaction relation for path patterns is $\vDash_A$, defining a node-isomorphism between the graph G and path pattern π. It is the same as $\vDash_T$, only now nodes must be distinct in a path p instead of edges. Hence, $(p, G, u) \vDash_A \pi$ for a path p and path pattern π is equal to the definition of $\vDash_W$ with the precondition that all nodes in p are distinct.

2.2.4 SIMPLE

The fourth, and last, satisfaction relation for path patterns is $\vDash_S$, also defining a node-isomorphism between the graph G and path pattern π, but allowing the first and last node to be the same. Hence, $(p, G, u) \vDash_A \pi$ for a path p and path pattern π is equal to the definition of $\vDash_W$ with the precondition that all nodes in p are distinct, except for the first and last node in a path, those may be the same node.

2.3 Pattern matching

Now, a path pattern π can be matched onto a graph G using one of the satisfaction relations from Table 2. Again, the definition used here for pattern matching is taken from Francis et al. [13] with a single deviation to allow for all four evaluation modes.

So, the set of free variables of a node pattern $\chi = (a, L, P)$, denoted by $\mathsf{free}(\chi)$, is $\{a\}$ whenever a is not nil, and empty otherwise. For a relationship pattern ρ, the set $\mathsf{free}(\rho)$ is defined equivalently. Now, for a path pattern π the set of free variables is defined as $\mathsf{free}(\pi)$ which is the union of all free variables of each node and relationship pattern occurring in π. Take for example the pattern π in Example 2.2, then $\mathsf{free}(\pi) = \{x, y\}$. For named patterns, $\mathsf{free}(\pi/a) = \mathsf{free}(\pi) \cup \{a\}$. Then, I obtain the following definition for a pattern matching.

Definition 2.3 (Pattern matching) *Let π be a path pattern, G a property graph, u an assignment and $\vDash$ one of the four satisfaction relations, then a matching of pattern π on property graph G using assignment u for a satisfaction relation $\vDash$ is defined as:*

$$\mathit{match}(\pi, G, u, \vDash) = \biguplus_{p \ in \ G} \left\{ u' \;\middle|\; \begin{array}{l} \mathit{dom}(u') = \mathit{free}(\pi) - \mathit{dom}(u) \\ \mathit{and}\ (p, G, u \cdot u') \vDash \pi \end{array} \right\} \quad (1)$$

Similar to Cypher, in 1, $\uplus$ stands for bag union, i.e. whenever a new combination of π and p is found such that $(p, G, u \cdot u') \vDash \pi$, a new occurrence of u' is added to $\mathit{match}(\pi, G, u, \vDash)$.

2.3.1 Example of pattern matching

In Section 2.2.1 two examples are given that satisfy the same paths in all four evaluation modes as no duplicate edges or nodes are matched. However, there is an eminent difference between the four satisfaction relations. This difference will be illustrated with the following example that is based on the graph in Fig. 3.

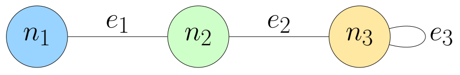


Figure 3: Simple example property graph G

Example 2.3 *Consider graph G in Fig. 3 and let π be the following rigid path pattern:*

```
(x)~[]~(y)~[]~(z)
```

Rewriting this to the defined mathematical representation, π becomes:

$$\underbrace{(x, \emptyset, \emptyset)}_{\chi_1}\underbrace{(-, \mathit{nil}, \emptyset, \emptyset, (1,1))}_{\rho_1}\underbrace{(y, \emptyset, \emptyset)}_{\chi_2},\underbrace{(-, \mathit{nil}, \emptyset, \emptyset, (1,1))}_{\rho_2}\underbrace{(z, \emptyset, \emptyset)}_{\chi_3}$$

It is easy to see that $\mathit{free}(\pi) = \mathit{free}(\chi_1) \cup \mathit{free}(\rho_1) \cup \mathit{free}(\chi_2) \cup \mathit{free}(\rho_2) \cup \mathit{free}(\chi_3) = \{x, y, z\}$, as only the node patterns have a name. Now, $\mathit{match}(\pi, G, u, \vDash)$ will be

Table 3: Output of pattern matching for each satisfaction relation with color-coding

Table 3a: $\vDash_W$

	x	y	z
1	n_1	n_2	n_1
2	n_1	n_2	n_3
3	n_2	n_1	n_2
4	n_2	n_3	n_2
5	n_2	n_3	n_3
6	n_3	n_2	n_1
7	n_3	n_2	n_3
8	n_3	n_3	n_2
9	n_3	n_3	n_3

Table 3b: $\vDash_S$

	x	y	z
1	n_1	n_2	n_1
2	n_1	n_2	n_3
3	n_2	n_1	n_2
4	n_2	n_3	n_2
5	n_3	n_2	n_1
6	n_3	n_2	n_3

Table 3c: $\vDash_A$

	x	y	z
1	n_1	n_2	n_3
2	n_3	n_2	n_1

Table 3d: $\vDash_T$

	x	y	z
1	n_1	n_2	n_3
2	n_2	n_3	n_3
3	n_3	n_2	n_1
4	n_3	n_3	n_2

compared for each satisfaction relation with assignment $u = \emptyset$. The computed assignments u' are shown in Table 3.

From this example it is clear that $\vDash_W$ allows for the most behaviour. For example, assignment 9 in Table 3a is not allowed for any other relations for example as it reuses node n_3 and edge e_3. All other assignments in Table 3a are allowed by at least one other satisfaction relation. Additionally, the slight difference between $\vDash_S$ and $\vDash_A$, where the first satisfaction relation does allow the first and last node to be equal, leads to much more allowed behaviour (i.e. compare the assignments in Table 3b and Table 3c).

2.4 Matching multiple patterns

GQL allows a match clause to contain a tuple of path patterns $\bar{\pi} = (\pi_1, \ldots, \pi_n)$, similar to a tuple of path patterns in Cypher [13]. In contrast to Cypher, each pattern π_i has its own corresponding satisfaction relation $\vDash_i$ as the evaluation mode is chosen in the path_pattern_prefix which is contained in the path pattern (see Fig. 2). Also, each path pattern π_i has its own assignment u_i. Hence, each pattern must be evaluated individually. Afterwards, the cross product of all results is computed. Then, the computed cross product is reduced if there are any equivalent variable names as for each such variable name the assignment to a node or edge must be unique in $\bar{\pi}$. So the following definition is obtained.

Definition 2.4 (Multiple patterns matching) *Let $M_i = \mathsf{match}(\pi_i, G, u_i, \vDash_i)$ be the set of assignments that output a match. Let $M = M_1 \times \cdots \times M_n$ be the cross product of all sets of assignments such that for each $1 \leq j < k \leq n$ we have for each $a \in \mathsf{dom}(u'_j) \cap \mathsf{dom}(u'_k)$ that $u'_j(a) = u'_k(a)$, i.e. a is assigned to the same node or edge. Then, M is the output of the match clause containing $\bar{\pi}$.*

Example 2.4 *To illustrate matching of multiple patterns, again consider graph G in Fig. 3 and let me have the following two match clauses:*

```
MATCH ACYCLIC (x)~[]~(y)~[]~(z) , TRAIL (x)~[]~(y)~[]~(z)
```

as Q_1 and

```
MATCH ACYCLIC (a)~[]~(b)~[]~(c) , TRAIL (x)~[]~(y)~[]~(z)
```

as Q_2.

For Q_1 I have $\bar{\pi} = (\pi_1, \pi_2)$ where $\pi_1 = \pi_2 = \chi_1\rho_1\chi_2\rho_2\chi_3$ as defined in Example 2.3, and $\vDash_1 = \vDash_A$ and $\vDash_2 = \vDash_T$. Now, for $u_1 = u_2 = \emptyset$ M_1 is the output seen in Table 3c and M_2 is the output seen in Table 3d. As $\mathsf{dom}(u'_1) = \mathsf{dom}(u'_2) = \{x, y, z\}$ only assignments are selected that are equal, i.e. the two assignments shown in Table 4a.

For Q_2 I have $\bar{\pi} = (\pi_3, \pi_4)$ where $\pi_4 = \pi_1$ and $\pi_3 = \chi_4\rho_1\chi_5\rho_2\chi_6$ where χ_4, χ_5 and χ_6 are the same as resp. χ_1, χ_2 and χ_3, only with names a, b and c. Again, the satisfaction relations $\vDash_3 = \vDash_A$ and $\vDash_4 = \vDash_T$. Now, for $u_3 = u_4 = \emptyset$ M_3 is the output seen in Table 3c only with columns a, b and c instead of x, y and z, and M_2 is the output seen in Table 3d. In this case $\mathsf{dom}(u'_3) \cap \mathsf{dom}(u'_4) = \emptyset$, so much more assignments are obtained as is visible in Table 4b.

Table 4: Output of multiple pattern matching

Table 4a: Output Q_1

	x	y	z
1	n_1	n_2	n_3
2	n_3	n_2	n_1

Table 4b: Output Q_2

	a	b	c	x	y	z
1	n_1	n_2	n_3	n_1	n_2	n_3
2	n_1	n_2	n_3	n_2	n_3	n_3
3	n_1	n_2	n_3	n_3	n_2	n_1
4	n_1	n_2	n_3	n_3	n_3	n_2
5	n_3	n_2	n_1	n_1	n_2	n_3
6	n_3	n_2	n_1	n_2	n_3	n_3
7	n_3	n_2	n_1	n_3	n_2	n_1
8	n_3	n_2	n_1	n_3	n_3	n_2

$$
\begin{array}{rcl}
\mathsf{expr} & ::= & v \mid a \mid \mathsf{expr}\,\texttt{.}\,k \qquad v \in \mathcal{V}, a \in \mathcal{A}, k \in \mathcal{K} \\
 & & \mid \mathsf{expr}\ \texttt{OR}\ \mathsf{expr} \mid \mathsf{expr}\ \texttt{AND}\ \mathsf{expr} \mid \mathsf{expr}\ \texttt{XOR}\ \mathsf{expr} \mid \texttt{NOT}\ \mathsf{expr} \\
 & & \mid \mathsf{expr}\ \texttt{IS}\ (\texttt{NOT})?\ \mathsf{truth\_value} \\
 & & \mid \mathsf{expr}\ \texttt{<}\ \mathsf{expr} \mid \mathsf{expr}\ \texttt{<=}\ \mathsf{expr} \mid \mathsf{expr}\ \texttt{>=}\ \mathsf{expr} \mid \mathsf{expr}\ \texttt{>}\ \mathsf{expr} \\
 & & \mid \mathsf{expr}\ \texttt{=}\ \mathsf{expr} \mid \mathsf{expr}\ \texttt{<>}\ \mathsf{expr} \\
\mathsf{graph\_expr} & ::= & a \qquad a \in \mathcal{A} \\
\mathsf{label\_expr} & ::= & \mathsf{label\_term}\ (\texttt{|}\ \mathsf{label\_term})* \\
\mathsf{label\_term} & ::= & \mathsf{label\_primary}\ (\texttt{\&}\ \mathsf{label\_primary})* \\
\mathsf{label\_primary} & ::= & \texttt{!}?\ (\ l \mid \texttt{\%} \mid \texttt{(}\ \mathsf{label\_expr}\ \texttt{)}\ ) \qquad l \in \mathcal{L} \\
\mathsf{truth\_value} & ::= & \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{UNKNOWN} \mid \texttt{NULL}
\end{array}
$$

Figure 4: Syntax of expressions

3 Complete syntax

Besides pattern matching, GQL consists of other key components as well. In this section I define the syntax of other key components, namely expressions, clauses and queries. Pattern matching, as defined in Section 2, expressions, clauses and queries constitute the desired formalization of the part of GQL that is within the scope of this internship.

In Fig. 2 the syntax of patterns is given. The syntax of expressions, which are derived from the token expr, is defined in Fig. 4. The syntax of queries, derived from the token query, is defined in Fig. 5. The syntax of a match clause, the only clause within the scope of the internship, is defined in Fig. 6. Notice that **WHERE** is not an actual clause, but its semantics is a table to table function.

Looking at the syntax of queries in Fig. 5, it is possible to leave out a set_quantifier in a return_statement or other_set_operator, and a set_quantifier or **MAX** in a union_operator. If a set_quantifier is left out in a return_statement, then implicitly the set quantifier **ALL** is assumed. If a set_quantifier (or **MAX**) is left out in a union_operator or other_set_operator, then implicitly the set quantifier **DISTINCT** is assumed.

4 Complete semantics

Now that the full syntax is defined in Section 3, I will now continue to lay out the semantics of GQL. I follow the standard convention for semantics, so the semantic function is denoted as $[\![\cdot]\!]$ and can contain an expression, query or condition. In contrast to the formal semantics for Cypher given by Francis et al.

```
              query ::=  query_expr (query_conjunction query_expr)*
  query_conjunction ::=  set_operator | OTHERWISE
         query_expr ::=  focused_query_expr | ambient_query_expr
 focused_query_expr ::=  (FROM s match_clause+)+ return_statement      s ∈ S
 ambient_query_expr ::=  match_clause+ return_statement
   return_statement ::=  RETURN set_quantifier? (* | return_list)
       set_operator ::=  union_operator | other_set_operator
     union_operator ::=  UNION (set_quantifier | MAX)?
 other_set_operator ::=  (EXCEPT | INTERSECT) set_quantifier?
     set_quantifier ::=  DISTINCT | ALL
        return_list ::=  return_item (, return_item)*
        return_item ::=  expr (AS a)?      a ∈ 𝒜
```

Figure 5: Syntax of queries

```
      match_clause ::=  (OPTIONAL | MANDATORY)? MATCH path_pattern_list where_clause?
 path_pattern_list ::=  path_pattern (, path_pattern)*
      where_clause ::=  WHERE expr
```

Figure 6: Syntax of clauses

[13], GQL allows the user to select a graph during execution of the query using the **FROM** clause. Hence, the semantics is defined by the input graph database D and a current working property graph G', which is denoted by $[\![\cdot]\!]_{D,G'}$.

4.1 Semantics of expressions

The semantics of an expression e is not only defined by a property graph G and graph database D, but also by an assignment u that binds the names used in e to a node or edge identifier in G. Hence, let $u = (a_1 : v_1, \ldots, a_n : v_n)$ be a fixed record that associates values $v_1, \ldots, v_n \in \mathcal{V}$ with full names $a_1, \ldots, a_n \in \mathcal{A}^2$. So the semantics of an expression e is $[\![e]\!]_{D,G',u}$. The details of the semantics are given below.

4.1.1 Values and variables

- $[\![v]\!]_{D,G',u} = v$
- $[\![a]\!]_{D,G',u} = u(a)$

4.1.2 Maps

Let $v \in V$

- $[\![e.k]\!]_{D,G',u} = \begin{cases} v & \text{if } [\![e]\!]_{D,G',u} \in N \cup E \text{ and } (k,v) \in \phi([\![e]\!]_{D,G',u}) \\ \texttt{unknown} & \text{if } [\![e]\!]_{D,G',u} \in N \cup E \text{ and } (k,v) \notin \phi([\![e]\!]_{D,G',u}) \\ \text{Not valid} & \text{otherwise} \end{cases}$

4.1.3 Logic

Let e and e' be expressions such that $[\![e]\!]_{D,G',u}, [\![e']\!]_{D,G',u} \in \{\texttt{true}, \texttt{false}, \texttt{unknown}\}$.

- $[\![e \texttt{ OR } e']\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = \texttt{true} \\ & \text{or } [\![e']\!]_{D,G',u} = \texttt{true} \\ \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = [\![e']\!]_{D,G',u} = \texttt{false} \\ \texttt{unknown} & \text{otherwise} \end{cases}$

- $[\![e \texttt{ AND } e']\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = [\![e']\!]_{D,G',u} = \texttt{true} \\ \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = \texttt{false} \\ & \text{or } [\![e']\!]_{D,G',u} = \texttt{false} \\ \texttt{unknown} & \text{otherwise} \end{cases}$

- $[\![e \texttt{ XOR } e']\!]_{D,G',u} = \begin{cases} \texttt{unknown} & \text{if } [\![e]\!]_{D,G',u} = \texttt{unknown} \\ & \text{or } [\![e']\!]_{D,G',u} = \texttt{unknown} \\ \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = [\![e']\!]_{D,G',u} \\ & \text{and } [\![e]\!]_{D,G',u} \neq \texttt{unknown} \\ \texttt{true} & \text{otherwise} \end{cases}$

- $[\![\texttt{NOT } e]\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = \texttt{false} \\ \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = \texttt{true} \\ \texttt{unknown} & \text{if } [\![e]\!]_{D,G',u} = \texttt{unknown} \end{cases}$

4.1.4 Value comparisons

Comparing values is an important part of query evaluation which impacts the evaluation drastically, for example an exception can be raised comparing two different data types or one could cast one data type to the other. In GQL there are multiple data types available, namely the base types in Section 1.1. However, not all base types are comparable to each other. The GQL ISO Standard

[21] specifies the comparison between two different *most specific types* not valid when those *most specific types* are not comparable. However, the GQL ISO Standard does not dive into much detail when two data types are comparable and if they can be casted to one another. For example, it states that "numbers are compared with respect to their algebraic value", where numbers comprise exact and approximate numeric data types. However, their algebraic value is somewhat vague. Besides, comparison of exact and approximate numeric data is not intuitive.

Take for example the exact numeric type with value 1.3 and the approximate numeric data type with value 1.3. Now, one may ask what the algebraic value of the approximated 1.3 will be as it is approximated by $1.30\ldots$ or $1.299\ldots$, hence, not exactly equal to 1.3. Hence, the user cannot anticipate on the approximate value being larger or smaller resulting in unpredictable behaviour. As the GQL ISO Standard does not shed too much light on comparison between numerical data types, it introduces unpredictably in a core part of GQL. Additionally, it is not the purpose of the project to define this equality in more depth. Therefore, I have decided to only compare values to its own base type.

Nulls: The semantics for comparison of null values is slightly different from SQL and Cypher. Instead of evaluating an expression and outputting *null*, GQL evaluates expressions to *unknown*. So, for example, the value of the expression is **unknown** if an argument is **null** in an expression. Still, the truth value of the keyword **NULL** is equivalent to **UNKNOWN**.

- $[\![e \ \mathsf{comp\_op}\ e']\!]_{D,G',u} = \texttt{unknown}$ if either $[\![e]\!]_{D,G',u} = \texttt{null}$ or $[\![e']\!]_{D,G',u} = \texttt{null}$, for $\mathsf{comp\_op} \in \{<, \leq, \geq, >, =, <>\}$.
- $[\![e \ \texttt{IS UNKNOWN}]\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = \texttt{unknown} \\ \texttt{false} & \text{if } [\![e]\!]_{D,G',u} \neq \texttt{unknown} \end{cases}$
- $[\![e \ \texttt{IS NOT UNKNOWN}]\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} \neq \texttt{unknown} \\ \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = \texttt{unknown} \end{cases}$

Truth values: Assume that $[\![e]\!]_{D,G',u} \in \{\texttt{true}, \texttt{false}, \texttt{unknown}\}$

- $[\![e \ \texttt{IS TRUE}]\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = \texttt{true} \\ \texttt{false} & \text{otherwise} \end{cases}$
- $[\![e \ \texttt{IS NOT TRUE}]\!]_{D,G',u} = \begin{cases} \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = \texttt{true} \\ \texttt{true} & \text{otherwise} \end{cases}$
- $[\![e \ \texttt{IS FALSE}]\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = \texttt{false} \\ \texttt{false} & \text{otherwise} \end{cases}$
- $[\![e \ \texttt{IS NOT FALSE}]\!]_{D,G',u} = \begin{cases} \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = \texttt{false} \\ \texttt{true} & \text{otherwise} \end{cases}$

Besides the **IS** operator, boolean values can also be compared using comparison operators $\{<, \leq, \geq, >, =, <>\}$ where we have that **true** is larger than **false**. Then, the values of all comparisons can be derived from the latter statement.

Base types: Assume that e and e' are expressions such that $[\![e]\!]_{D,G',u}$ and $[\![e']\!]_{D,G',u}$ are of the same base type.

- $[\![e = e']\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = [\![e']\!]_{D,G',u} \\ \texttt{false} & \text{otherwise} \end{cases}$

- $[\![e <> e']\!]_{D,G',u} = \begin{cases} \texttt{false} & \text{if } [\![e]\!]_{D,G',u} = [\![e']\!]_{D,G',u} \\ \texttt{true} & \text{otherwise} \end{cases}$

Additionally, all other comparison operators, $\{<, \leq, \geq, >\}$, are defined as well. Most importantly, rational numbers ($\mathbb{R}$) are compared on their exact value, floating point numbers ($\mathbb{F}$) are compared following the IEEE754 standard [8] and strings on their lexographic ordering following the Unicode Collocation Algorithm [7]. Other base types such as datetimes and durations also have a defined ordering, but are not relevant in the scope of this internship.

If the two base types of e and e' are not the same, then they are not comparable and an exception will be raised. Although this may not be the exact definition of comparison intended by the GQL ISO Standard [21], it definitely is bounds the interpreter to behaviour that is allowed. Therefore, the interpreter is coherent with the GQL ISO Standard.

Identifiers: Assume that e and e' are expressions such that $[\![e]\!]_{D,G',u}$ and $[\![e']\!]_{D,G',u}$ are both in the set of node identifiers N or both in the set of edge identifiers E of property graph G.

- $[\![e = e']\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = [\![e']\!]_{D,G',u} \\ \texttt{false} & \text{otherwise} \end{cases}$

Paths: Assume that e and e' are expressions such that $[\![e]\!]_{D,G',u}$ and $[\![e']\!]_{D,G',u}$ are both path values.

- $[\![e = e']\!]_{D,G',u} = \begin{cases} \texttt{true} & \text{if } [\![e]\!]_{D,G',u} = [\![e']\!]_{D,G',u} \\ \texttt{false} & \text{otherwise} \end{cases}$

4.2 Semantics of queries

A GQL query is a sequence of match clauses on (un)specified graphs and a return statement. Additionally, two queries can be unioned, intersected, the difference can be taken, or the second query can serve as backup if the first query does not find any matches. Hence, GQL allows much more behaviour than Cypher as Cypher does not have the capability to execute match clauses on specific graphs and only has the union operator [13]. The specification of a graph is written as a **FROM** clause, just like in SQL [17]. The **RETURN** statement in GQL is the same as in Cypher, i.e. its return item list is either $*$ (token for returning all variable bindings), or a sequence of possibly named expressions.

Additionally, GQL also allows for **SELECT** statements (used in SQL and similar to **RETURN** statements [17]). However, the **SELECT** statement is not discussed in this internship as the evaluation of a **SELECT** statement is not yet defined. Besides, it is suggested at this point that it is best implemented via syntax transformation to a **RETURN** statement.

Now, in order to express the semantics of queries, let α be an injective function that maps expressions to names. Although GQL is a property graph query language, the semantics of queries and clauses on a property graph G is a function from binding tables to binding tables. Hence, the semantics of a query on property graph G from a binding table T is defined as $[\![\mathsf{query}]\!]_{D,G'}(T)$.

In order to combine multiple query outputs via union, intersection or difference, bag and set semantics should be applied. Hence, $\biguplus, \Cap, -$ are used to define bag union, bag intersection and bag difference and $\bigcup, \bigcap, \setminus$ are the set union, set intersection and set difference. Note that the only difference is the possibility to have duplicate elements using bag semantics, where set semantics can only have zero or one occurrence of an element. In the semantics of GQL only bag semantics is used, and duplicates are eliminated using function δ (see Section 1.4).

4.2.1 Return

As the **RETURN** statement in GQL is the same as in Cypher, I will use the semantics of Francis et al. [13] for the **RETURN** statement. Hence, assume that the fields of T are $b_1, \dots, b_q$. Also assume, if we have a return list e_1 [**AS** a_1], $\dots, e_m$ [**AS** a_m], then $a'_i = a_i$ if **AS** a_i is present in the list, and $a'_i = \alpha(e_i)$ otherwise, with the added requirement that all the a'_is are distinct, i.e. there are no ambiguous names.

- $[\![\texttt{RETURN ALL } *]\!]_{D,G'}(T) = T$
- $[\![\texttt{RETURN ALL } e_1 \ [\texttt{AS } a_1], \dots, e_m \ [\texttt{AS } a_m]]\!]_{D,G'}(T) =$
$$\biguplus_{u \in T} \{(a'_1 : [\![e_1]\!]_{D,G',u}, \dots, a'_m : [\![e_1]\!]_{D,G',u})$$
- $[\![\texttt{RETURN DISTINCT} \dots]\!]_{D,G'}(T) = \delta([\![\texttt{RETURN ALL } \dots]\!]_{D,G'}(T))$

4.2.2 Intersection

Let Q_1, Q_2 be queries.

- $[\![Q_1 \texttt{ INTERSECT ALL } Q_2]\!]_{D,G'}(T) = [\![Q_1]\!]_{D,G'}(T) \Cap [\![Q_2]\!]_{D,G'}(T)$
- $[\![Q_1 \texttt{ INTERSECT DISTINCT } Q_2]\!]_{D,G'}(T) =$
$$\delta([\![Q_1 \texttt{ INTERSECT ALL } Q_2]\!]_{D,G'}(T))$$

4.2.3 Union

Let Q_1, Q_2 be queries.

- $[\![Q_1 \texttt{ UNION ALL } Q_2]\!]_{D,G'}(T) = [\![Q_1]\!]_{D,G'}(T) \uplus [\![Q_2]\!]_{D,G'}(T)$
- $[\![Q_1 \texttt{ UNION DISTINCT } Q_2]\!]_{D,G'}(T) = \delta([\![Q_1 \texttt{ UNION ALL } Q_2]\!]_{D,G'}(T))$
- $[\![Q_1 \texttt{ UNION MAX } Q_2]\!]_{D,G'}(T) = [\![Q_1 \texttt{ UNION ALL } Q_2]\!]_{D,G'}(T) - [\![Q_1 \texttt{ INTERSECT ALL } Q_2]\!]_{D,G'}(T)$

4.2.4 Difference

Let Q_1, Q_2 be queries.

- $[\![Q_1 \texttt{ EXCEPT ALL } Q_2]\!]_{D,G'}(T) = [\![Q_1]\!]_{D,G'}(T) - [\![Q_2]\!]_{D,G'}(T)$
- $[\![Q_1 \texttt{ EXCEPT DISTINCT } Q_2]\!]_{D,G'}(T) = \delta([\![Q_1]\!]_{D,G'}(T)) - [\![Q_2]\!]_{D,G'}(T)$

4.2.5 `OTHERWISE`

Let Q_1, Q_2 be queries.

- $[\![Q_1 \texttt{ OTHERWISE } Q_2]\!]_{D,G'}(T) = \begin{cases} [\![Q_1]\!]_{D,G'}(T) & \text{if } [\![Q_1]\!]_{D,G'}(T) \text{ contains at least one record} \\ [\![Q_2]\!]_{D,G'}(T) & \text{otherwise} \end{cases}$

4.3 Semantics of clauses

The last component to complete the part of GQL within the scope of this internship is the semantics of clauses. Within the scope I only define the match clause and the where clause, although the latter is not necessarily a clause as it must interact with a match clause. Again, the syntax is very similar to Cypher, only the keyword **`MANDATORY`** can be used as well at the place of **`OPTIONAL`** and a **`FROM`** clause can change the current working graph. Therefore, the semantics of Francis et al. will be used with a slight adjustment for **`MANDATORY`** statements. Notice that the most crucial semantics is pattern matching using $\mathsf{match}(\bar{\pi}, G, u)$ as described in Definition 2.3 Equation 1.

As mentioned in Section 4.2, the semantics of clauses is a function from binding tables to binding tables. Moreover, as defined in Section 2.3, a match clause adds field names to T that coincide to names in the pattern but are not in u. Additionally, using the bag union it adds matched tuples to T. Also, a where clause filters records according to the evaluation of an expression.

4.3.1 Where clause

Let e be an expression.

- $[\![\texttt{WHERE}\ e]\!]_{D,G'}(T) = \{u \in T \mid [\![e]\!]_{D,G',u} = \texttt{true}\}$

4.3.2 Matching clause

Let $s \in S$ be a graph name, $\bar{\pi}$ be a tuple of path patterns, $\vDash$ a tuple of satisfaction relations and e the output of an expression. Also, $\times$ is now the Cartesian product between two records.

- $[\![\texttt{FROM}\ s\ \mathsf{match\_clause}]\!]_{D,G'} = [\![\mathsf{match\_clause}]\!]_{D,D(s)}$, i.e. set the current working graph to $D(s)$
- $[\![\texttt{MATCH}\ \bar{\pi}]\!]_{D,G'}(T) = \biguplus_{u \in T} \{u \times u' \mid u' \in \mathsf{match}(\bar{\pi}, G', u, \vDash)\}$
- $[\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_{D,G'}(T) = [\![\texttt{WHERE}\ e]\!]_{D,G'}\Big([\![\texttt{MATCH}\ \bar{\pi}]\!]_{D,G'}(T)\Big)$
- $[\![\texttt{OPTIONAL MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_{D,G'}(T) =$
$$\begin{cases} [\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_{D,G'}(T) & \text{if } [\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_{D,G'}(T) \neq \emptyset \\ \biguplus_{u \in T} (u, (\mathsf{free}(u, \bar{\pi}) : \texttt{null})) & \text{otherwise} \end{cases}$$
- $[\![\texttt{OPTIONAL MATCH}\ \bar{\pi}]\!]_{D,G'}(T) =$
$$[\![\texttt{OPTIONAL MATCH}\ \bar{\pi}\ \texttt{WHERE true}]\!]_{D,G'}(T)$$
- $[\![\texttt{MANDATORY MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_{D,G'}(T) =$
$$\begin{cases} [\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_{D,G'}(T) & \text{if } [\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ e]\!]_{D,G'}(T) \neq \emptyset \\ \text{Not valid} & \text{otherwise} \end{cases}$$
- $[\![\texttt{MANDATORY MATCH}\ \bar{\pi}]\!]_{D,G'}(T) =$
$$[\![\texttt{MANDATORY MATCH}\ \bar{\pi}\ \texttt{WHERE true}]\!]_{D,G'}(T)$$

5 Interpreter

In the previous sections I have created a static formalization of a part of GQL. Using the defined formalization I will develop a naive implementation of a GQL interpreter that can process GQL queries. Such an implementation will contribute to the understanding of GQL by peers and also show how the defined formalization can be used in practice. Currently, there is no working implementation of GQL as the GQL ISO Standard [21] is not yet finished, hence, it can also be seen as an exploration tool for future users or implementers of GQL.

Evidently, there is no machine available at this point in time that can directly evaluate GQL queries based on the defined syntax. Therefore, the interpreter needs to be written from scratch which leaves multiple approaches that can be chosen. In this section I will discuss each component needed to build an interpreter and address why I favored this approach above others.

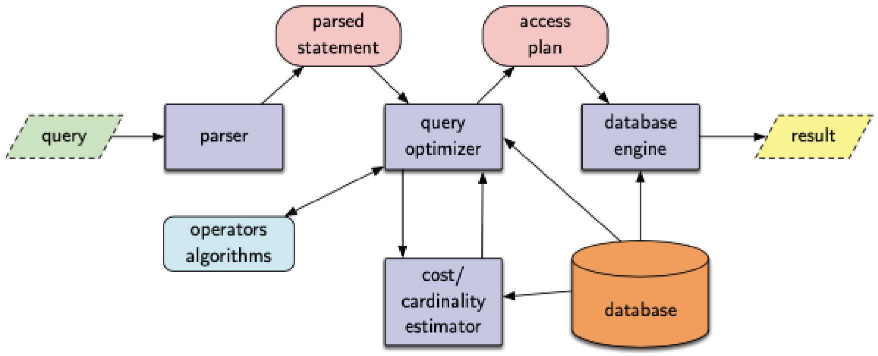


Figure 7: Standard query processing pipeline by Bonifati et al. [11]

In order to get a grasp of the system overview, look at the standard query processing pipeline in Fig. 7 described by Bonifati et al. [11]. The process starts from the left with a query, which is parsed to an input that the query optimizer can read. From there the cardinality of a query may be estimated to optimize a query plan, i.e. define the evaluation order of the query. When such an access plan is eventually produced, the database engine gathers all data from the database and returns the result in the requested format. As this internship is not focused on query optimization, but merely focused on exploring GQL I will only implement a subset of the pipeline, see Fig. 8.

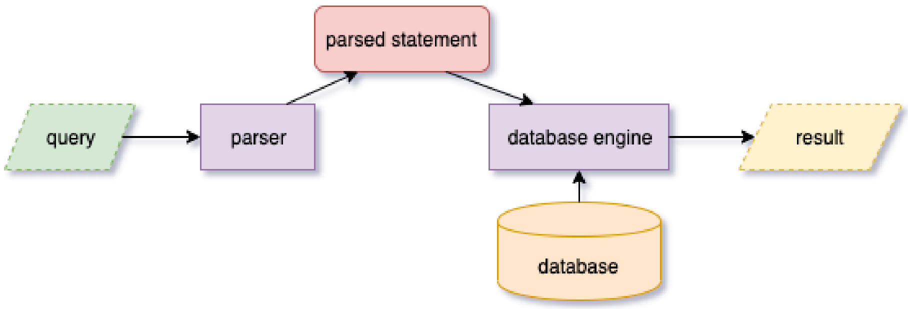


Figure 8: Simplified query processing pipeline

The interpreter will be written in Java 8 and I will use ANTLR v4 [19] to parse queries and JSON files as database. Using ANTLR v4 the database engine uses the Visitor pattern to evaluate the query to a result which is written to the console. A side effect of the evaluation is a piece of Latex describing the formalization of the query. In the remainder of this section I will elaborate on each element of the simplified query processing pipeline shown in Fig. 8.

5.1 Query

The first step in evaluating a query is of course obtaining the query itself from the user. I have chosen for two different ways to supply a GQL query. The first one is writing a query in the console and executing the query after typing *execute*. The other is to write a query into a *.gql* file and give the filepath to the query. Both methods allow for the same set of queries, where the first method can be used for direct querying, and the latter for saving queries on your computer and re-executing a saved query.

5.2 Parser

Now that a query is obtained, it needs to be read by the computer. Of course a computer cannot do this by itself, here is where a parser comes into play. More specifically, a parser will read all characters from the input, construct tokens from the input characters using a lexer and eventually create a syntax tree (also named parse tree), see Fig. 9, that a computer can evaluate. Luckily, this process is automated by multiple different tools decreasing the complexity of the eventual interpreter.

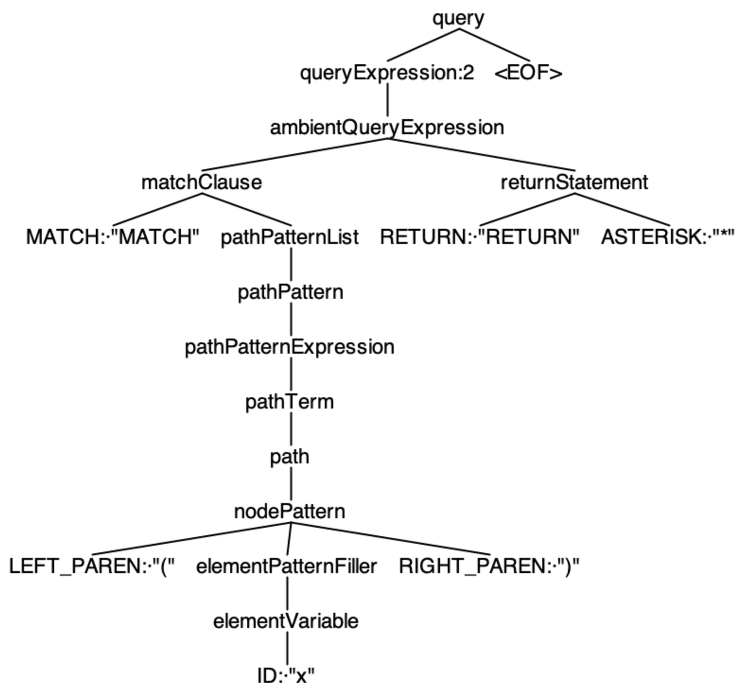


Figure 9: Parse tree of query `MATCH` (x) `RETURN` *

However, before I dive into the chosen parser strategy, a parser must know how to read an input from a specific language. Equivalently to how humans read, a parser will also need a grammar to which the language must adhere. As a matter of fact, I have already proposed the grammar that will be used by the parser, namely the syntax defined in Fig. 2, 4, 5, and 6. In order for the parser to read GQL queries, the defined syntax is brought together in a lexer and grammar file. The lexer file contains keywords, such as **MATCH** and **RETURN**, and definitions of literals, such as strings, floats and integers. The grammar contains the syntax defined in this report. Using both the lexer and grammar together and an input query you can construct parse trees that can be evaluated by a computer.

Hereafter, it is possible to either write a parser by hand or use a parser generator like ANTLR [19] or Bison [3]. As the project is focused on a naive implementation that must enhance the understanding of GQL and is not yet a complete implementation, it is clearly not focused on performance or incremental parsing. Besides, parser generators are already out there since the 70s [18] and are developed since. Not only are parser generators developed over the past decades, they are used in industry and academia as well. Twitter search, NetBeans IDE and even Oracle's SQL Developer IDE use ANTLR for parsing [20].

Hence, I have chosen ANTLR as framework to develop a GQL interpreter as it a prominent parser generator that enables custom evaluation as well. By its nature, ANTLR generates a parser from the constructed lexer and grammar file that will construct a syntax tree as parsed statement from a given input, an example of such a tree can be seen in Fig. 9. Additionally, ANTLR recognizes syntax errors before evaluation and also has the possibility of custom error handling [20]. Henceforth, after such a parse tree is constructed correctly, it can be evaluated by the database engine to obtain a result.

5.3 Parsed statement

Thus a syntax tree is now given as parsed statement. Looking at Fig. 9 you can see multiple terms in the syntax tree that are in the defined syntax as well. In fact, the root of the syntax tree can be constructed from query in Fig. 5. Specifically, the query expression only contains a **MATCH** clause, so corresponds to the ambient_query in Fig. 5. Now that all elements of the query are recognised according to the grammar, I will show how the database engine will evaluate the given query.

5.4 Database engine

After receiving the syntax tree from the parser the database engine will evaluate the given query on a graph obtained from the database and provide the user the output. Therefore, the main objective of the database engine is to walk over the parse tree and interpret what it sees. As ANTLR is written in Java, the database engine is also restricted to Java. Fortunately, Java is a well developed OO programming language used universally. At this point I have selected one of two strategies of evaluating queries, i.e. I have chosen to implement the graph computing framework from scratch over rewriting queries to an existing graph query language such as Apache Tinkerpop Gremlin [2], Cypher [4] or SPARQL [6].

The latter option would mean that a mapping from the given formal semantics for GQL to an existing formal semantics for the other language needed to be given. More importantly, that mapping should attain equivalency between the output of queries in GQL to their image in the other language. However, each language diverges too much from GQL in at least one aspect, which can be found in Table 5.

	GQL	Cypher	Gremlin	SPARQL
Evaluation in Java	×	✓	✓	✓
Ternary logic as in GQL	✓	✓	×	×
Homomorphic subgraph matching	✓	×	✓	✓
Edge-isomorphic subgraph matching	✓	✓	×	×
Node-isomorphic subgraph matching	✓	×	×	×

Table 5: Query languages and their features

Evaluation in Java is possible for Gremlin which has an endpoint for Cypher as well, and SPARQL queries can be evaluated in Java using Apache Jena's ARQ [1]. As GQL is not yet implemented anywhere, it is definitely not available in Java. The second aspect, ternary logic is important for equivalence of data in queries. Gremlin and SPARQL do not support `NULL` values, hence, they do not support three-valued logic, whilst GQL and Cypher do. Although the difference is subtle, it may lead to complex query rewrites and even outcomes in GQL that are invalid in SPARQL or Gremlin.

Above all, GQL is defined to support subgraph matching with different morphisms. From Example 2.3 it is clear that edge and node isomorphisms can be obtained from a homomorphism, however, this is not true the other way around. Hence, Cypher seriously lacks capabilities here and is definitely not suitable to rewrite queries in. Both SPARQL and Gremlin do not support `NULL` values, so is not suitable either as `IS NULL` statements are just not valid in Gremlin and SPARQL.

Therefore, I have decided to write the database engine from scratch. This way I have full control over data types, their equality, ternary logic, join strategies, and so forth. Additionally, the parser is still open for extension as I only discuss a fragment of GQL, hence, implementing the database engine from scratch does not impose restrictions on further extensions of the interpreter.

In order to fully evaluate the syntax tree given by ANTLR, ANTLR provides a tree walker via the Visitor pattern [15]. Using the Visitor pattern I construct values, expressions, node and edge patterns, and path patterns. After I have obtained a path pattern, I find the match on a graph, possibly filter it by a where clause, or impose a mandatory, optional or distinct statement. If there is a query conjunction, I obtain the left and right side, after which I execute the query conjunction. Multiple query conjunctions are evaluated from left to right.

There are different approaches to speeding up evaluation of queries, such as stopping when the intermediate output is empty or only evaluate later segments of the query on intermediate results. However, such optimizations can influence the output of two queries that are equivalent in their formal semantics. A simple example is the union of two queries, resp. $Q1$ and $Q2$, where $Q1$ will result in an empty output and $Q2$ in a runtime exception. Evaluating the union of both queries from left to right and stopping when a intermediate result is empty will lead to two semantically equal queries with different evaluation results in the interpreter. $Q1$ `UNION` $Q2$ will evaluate to an empty result, whereas $Q2$ `UNION` $Q1$ will evaluate to an exception, whilst both queries are semantically equivalent. To prohibit such inconsistencies from happening the interpreter will not have any optimization methods and will evaluate the entire query from left to right. Therefore, the interpreter will always evaluate a given query consistently with right to its semantics unless multiple exceptions are thrown in different subqueries.

5.5 Database

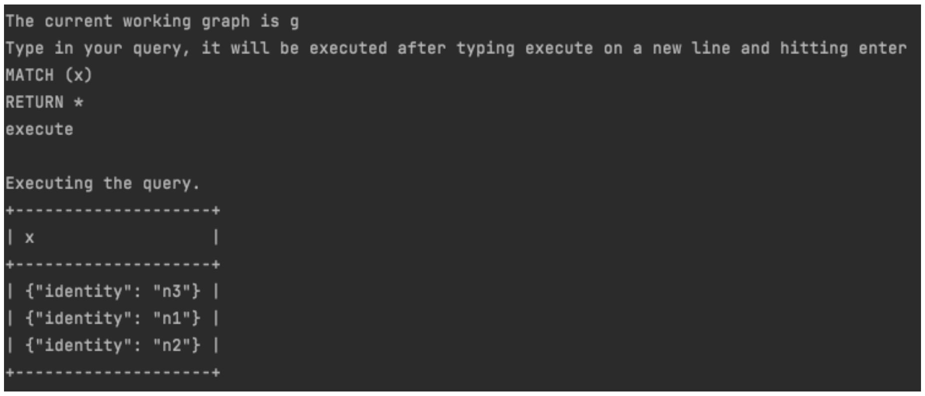
The evaluation of a GQL query must be evaluated over a property graph. The GQL ISO Standard [21] proposes to evaluate queries over a single working graph. As the implementation is build from scratch, I have designed a simple JSON format for nodes and edges, where nodes and edges are in two separate files (see Appendix C). The interpreter loads the graph into main memory using the same lexer that is used for parsing the GQL query and a generated parser using ANTLR [19]. This way all data types are coherent across the database and the query input.

Additionally, using an in-memory database is faster than using a traditional database. Also, all graphs are persisted in the JSON files and loaded into memory at run-time. As only `FROM-MATCH-RETURN-WHERE` queries are within the scope of the project, it is not possible to change graphs at runtime. Furthermore, allowing queries over datasets that exceed the size of main memory is not the aim of the project, therefore an in-memory database without any serialization will suffice.

In the database each graph is named by the directory name the node and edge files are in. Hence, a focused match clause can be executed on a specific graph. If an ambient query is executed and no working graph is yet set, the first folder, assuming alphabetic order, is chosen.

5.6 Result

Subsequently to the evaluation of the query, a result will be presented to the user. Moreover, the result of the interpreter will be twofold. The first result is common for every database, namely the evaluation result of the query which will be presented in a binding table. The second result is added for exploratory



```
The current working graph is g
Type in your query, it will be executed after typing execute on a new line and hitting enter
MATCH (x)
RETURN *
execute

Executing the query.
+--------------------+
| x                  |
+--------------------+
| {"identity": "n3"} |
| {"identity": "n1"} |
| {"identity": "n2"} |
+--------------------+
```

Figure 10: Console output of query **MATCH** (x) **RETURN** * on graph G

or more formal use, namely a LaTeX style string that will contain the formal semantics of the query.

Although GQL is focused on property graphs, the evaluation result of a query will be given in a binding table (see Section 1.4). Presently, the evaluation result will be printed to the console of the user and is, as expected, composed via the **RETURN** statement. In this fashion the user can inspect the evaluation result directly, and even copy the output to a format of its preference. In Fig. 10 one can see the output of query **MATCH** (x) **RETURN** * (same query used for Fig. 9) on graph G shown in Fig. 3.

Furthermore, the semantic formalization of the input query will be presented to the user. Presenting this formalization will contribute to the understanding of queries in GQL, but also allow the user to reason about its equivalency to other queries. I have chosen to output the formalization in LaTeX as the given semantics contain mathematical symbols that are not easily presented in the console or other formats. Besides, LaTeX is a widely used formatting language in the academic world and widely available.

6 Discussion

During the course of my internship I have codified a formal algebra for read statements in GQL. Additionally, I implemented a first naive interpreter for GQL that will yield the evaluation result as well as the formalization of the query. The interpreter can be used to explore a part of its syntax, experiment with queries and produce formalizations of queries.

First of all, the formal algebra for GQL defined in this report is a first exploration of read statements of the new query language named GQL. Currently, GQL is not yet fully standardized and definitely not implemented in an existing database. Therefore, this first exploration is an academical instrument for scholars, but also industry, to understand the capabilities and architecture of

GQL.

Secondly, the formal algebra serves as comparison mechanism between queries and as mathematical definition of the evaluation process of GQL. The formal semantics demonstrates the evaluation of a GQL query, but also facilitates reasoning about equivalence of distinct queries. In contrast to the GQL ISO Standard [21], the formalization that is proposed abstracts from the implementation of GQL, presenting a universal and implementation-independent overview of the evaluation of a read statement in GQL that is concise. Moreover, the formalization may unravel or prove optimization strategies in the future.

As proposed by Guagliardo et al., a formal semantics must adhere to "real languages that people use" [17]. Although that was not possible presently due to the lack of a working implementation, the formal semantics and interpreter are coherent with the GQL ISO Standard [21]. Currently, SQL RDBMSs are known to stay close to the SQL ISO Standard [17], hence, I assume that upcoming implementations of the GQL ISO Standard will stay close as well. Plus, the formal semantics can be adapted or extended to align with an implementation that varies a little.

Whereas Guagliardo et al. provide an experimental implementation that is validated using generated queries where their evaluation results are compared to evaluation results of existing RDBMSs, for GQL that is not possible due to the lack of any implementation. Still, my implementation adheres to the GQL ISO Standard [21] which should be the foundation for any implementation of GQL, hence, contributing to the validation of upcoming systems. Additionally, my implementation is capable of translating a GQL query into the formal semantics, therefore, contributing to the formalization of GQL for read statements.

Simultaneously, there is a broad field of future research that will become prevalent around GQL in the upcoming years. Therefore, I have divided suggestions for further investigation into three parts, i.e. topics of interest for language designers, academics and system designers respectively in Section 6.1, 6.2 and 6.3.

6.1 Language designer interests

Evidently, the given formal semantics does not cover the entire GQL query language. Thereupon, a logical extension of my research would be to include other parts of the GQL query language as well. Facets like variable length path pattern matching, ordering, grouping, aggregation, schema definitions, constraints and updates deserve to be added to the current formal semantics.

Besides, the given formalization and interpreter are fully based on the GQL ISO Standard [21]. Therefore, it might be of use to examine the given formalization and interpreter in order to see whether it is fully aligned with the desired goal of the GQL ISO Standard.

Also, I have assured that the given formalization is a subset of the entire specification of GQL, therefore, I have left out path union and multiset alternation from the syntax. Following the GQL ISO Standard, the latter is rewritten to a path union and it was not clear how a path union should be evaluated.

Specifically, if the domains of two paths are $\{a, b\}$ and $\{c\}$, I could not fully understand how two binding tables with those domains containing the matches of the two paths must be unioned. Still, this part of the syntax is intriguing as it is not present in any other existing query language, hence, deserving future research.

6.2 Academic interests

Another topic of interest that is entitled to some attention is three-valued logic that is currently included in GQL. At a first glance, the proof of Guagliardo et al. [17] that "SQL queries have the same expressiveness under the three-valued and the two-valued semantics" seems to be generalizable towards the given formal semantics of GQL. As three-valued logic received criticism in SQL, it might be of interest to not include three-valued logic in GQL. Moreover, the main justification by Guagliardo et al. [17] to not remove three-valued logic from SQL is legacy code, however, GQL does not have legacy code. Still, it should be confirmed that this will cause a significant benefit, i.e. that people tend to think in terms of true and false only, hence, will enhance intuitive use of GQL, or that it enhances query optimization.

6.3 System designer interests

Beyond the research in semantics, multiple steps of the standard query processing pipeline (see Fig. 7) will demand research. Query optimization for GQL will definitely need attention in the upcoming years. GQL is the first graph database query language that will allow a query to contain different morphisms, which can be optimized differently. Additionally, database vendors will most likely implement at least a fraction of GQL, thus, look for methods that will optimize query evaluation.

Likewise, the cardinality estimator must extend to multiple types of morphisms in a single query and the schema structure of a property graph in GQL. Similarly, graph serialization, graph indices for access plans and join operators must be adapted to the property graph definition of GQL. Such future work will extend the given formal definition of a property graph in this report and extend the given syntax towards creating, updating and deleting property graphs.

Also, more elementary components are of interest. The given interpreter only accepts specific *.json* files as database input, whereas most data is not yet in this form. Therefore, other input formats enhance the use cases of the interpreter. Equally important, the result is now restricted to binding tables printed in the console. Naturally, at least one export format is desired to extract the requested data from the input data. This way one can share or save the output of a query.

7 Conclusion

All in all, this research is one of the first academic perspectives on a new query language that is not yet fully born. The objective of my research was to define a formal semantics for GQL and provide a naive implementation that will help understand the execution of a query. Because formalizing GQL fully is a colossal project, I have focused on reading statements, namely **`FROM-MATCH-RETURN-WHERE`** queries, only containing rigid graph patterns.

Before diving into the actual syntax and semantics of GQL, in Section 1 I have defined the mathematical structures on which the semantics is defined. First, a straightforward definition of values is given. From there a definition of a property graph, based on the property graph by Angles [9], is established together with an illustration of its different forms. Upon which a graph database is defined which allows an user to select a property graph by its name. At last, the structure for the output is given, namely a binding table.

Next, the syntax of patterns is given in Section 2.1 together with its mathematical representation. In Section 2.2 the satisfaction relations for four different morhpisms are characterized. Subsequently, pattern matching with a rigid graph pattern π on a graph G is defined and illustrated. Not only a definition for matching a single graph pattern is presented, also, a definition for matching multiple patterns is given. Now it is possible to obtain data from a property graph using a GQL match statement.

Furthermore, in Section 3 I present the full syntax of read statements in GQL that fits within the scope. Moreover, in Section 4 all semantics corresponding to the given syntax is presented in a similar manner as the semantics given by Francis et al. [13]. Starting from the semantics of expressions, then giving the semantics of queries, completing the definitions with the semantics of match and where clauses. Utilizing this semantics one can infer equality between queries and reason if the given evaluation result is mathematically correct.

Since GQL is a new query language without any implementations, a naive interpreter is implemented following the simple query pipeline outlined in Fig. 8. The interpreter will obtain a query from the user which it will parse using the defined syntax. Via the syntax generator ANTLR [19], a parse tree is constructed from which a tree walker constructs objects that match the formal semantics. Evaluating the query naively from left to right, the database engine will extract the requested data from the database and yield the desired result in a binding table as well as the formalization of the query. This allows one to experiment with GQL in two ways. First, to see how a query is evaluated and what the capabilities are of GQL. Secondly, compare queries via their formalization and look for possible optimizations.

Finally, there is enough room for future research as GQL is so young. First of all, the semantics and the interpreter can be extended to encompass all capabilities of GQL. Secondly, language designers can explore new components of GQL that are not available in any other language, like path union or evaluation matches using different graph morphisms. From an academic perspective, it is interesting to extend the research into ternary logic used by SQL and GQL as already proposed by Guagliardo et al. [17]. At last, system designers will look for optimizations of query evaluation in GQL, but also graph indexing and serialization are topics of interest.

All things considered, GQL is a new promising query language for property graphs that will expand and unify the current field of querying property graphs. By defining a formal semantics and bringing it to life in a working implementation, I have contributed an instrument for academics and industry to explore the future of graph query languages.

References

[1] Apache Jena - ARQ - A SPARQL Processor for Jena, . URL https://jena.apache.org/documentation/query/index.html.

[2] Apache TinkerPop, . URL https://tinkerpop.apache.org/.

[3] Bison - GNU Project - Free Software Foundation. URL https://www.gnu.org/software/bison/.

[4] Cypher Query Language - Developer Guides. URL https://neo4j.com/developer/cypher/.

[5] OlofMorra/GQL-parser: A GQL parser build with Antlr v4. URL https://github.com/OlofMorra/GQL-parser.

[6] SPARQL 1.1 Query Language. URL https://www.w3.org/TR/sparql11-query/.

[7] UTS #10: Unicode Collation Algorithm. URL https://www.unicode.org/reports/tr10/.

[8] ISO/IEC 10967-1:2012(E): Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic, 2012. URL https://standards.iso.org/ittf/PubliclyAvailableStandards/c051317_ISO_IEC_10967-1_2012.zip.

[9] R. Angles. The Property Graph Database Model. In *AMW*, 2018.

[10] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, and others. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1421–1432, 2018.

[11] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. *Querying graphs.* ynthesis Lectures on Data Management, 2018. ISBN 1681734311.

[12] G. D. Ciocîrdel and A. Uta. A G-CORE (Graph Query Language) Interpreter. Technical report, 2018.

[13] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer, and others. Formal semantics of the language cypher. *arXiv preprint arXiv:1802.09984*, 2018.

[14] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, volume 707 LNCS, chapter 5. Springer, Berlin, Heidelberg, 1993. doi: 10.1007/3-540-47910-4{_}21. URL https://link.springer.com/chapter/10.1007/3-540-47910-4_21.

[16] A. Green. GQL Manifesto, 2018. URL https://gql.today/.

[17] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *Proceedings of the VLDB Endowment*, 11(1):27–39, 2017.

[18] S. C. Johnson and R. Sethi. Yacc: a parser generator. *UNIX Vol. II: research system*, pages 347–374, 1990.

[19] T. Parr. ANTLR. URL https://www.antlr.org/.

[20] T. Parr. *The Definitive ANTLR 4 Reference* . The Pragmatic Programmers, LLC., Dallas, Texas, 1 edition, 1 2013. ISBN 978-1-93435-699-9. URL https://books.google.nl/books?hl=en&lr=&id=gA9QDwAAQBAJ&oi=fnd&pg=PT10&dq=parser+generator+antlr&ots=jqgmm6h670&sig=WzNEmJmKWDCihqKCWgv4imizWME&redir_esc=y#v=onepage&q=parser%20generator%20antlr&f=false.

[21] S. Plantikow and S. Cannan. Information technology - Database languages - GQL, 2021.

[22] N. Populi. What is the Most Widely-used Graph Query Language in 2018?, 2018. URL https://leapgraph.com/graph-query-languages/.

A Rewriting queries

The syntax defined in Fig. 2, 4, 5, and 6 is not fully implemented due to time limit of the internship or significant increases of complexity of implementation in contrast to rewriting a query. In this section of the appendix I exemplify how one could rewrite a query that fits within the defined syntax, but is not implemented. There are three such situations, namely label negations, nested label expressions or where clauses in elt_predicate.

A.1 Label negation

A label negation can be written in label_primary found in Fig. 4. Namely, a label negation is written as ! followed by a label $l \in \mathcal{L}$, label wildcard % or a label_expr. Now, a label negation cannot be added to a node pattern as negations for labels are not defined in here. However, such a negation works quite intuitively as it merely states that the element pattern may not contain that label. Now, let us see an example query, Q_1:

```
MATCH (a:!labelName)
RETURN *
```

Now, assume that $labelName \in \mathcal{L}$, then Q_1 returns each node n of a graph which does not contain the label $labelName$, or more formally, each node n for which $labelName \notin \lambda_N(n)$. This is logically equivalent to not returning each node n' that contains the label $labelName$ if the same name is used. This suggests that the following query Q_2 is equivalent:

```
MATCH (a)
RETURN *
EXCEPT
MATCH (a:labelName)
RETURN *
```

It is obvious that all nodes of a graph are returned by the first query expression, the second query expression returns all nodes containing label $labelName$ and the set operator **EXCEPT** makes sure that all nodes containing label $labelName$ are not returned, i.e. the expected output for a label negation.

A.2 Nested label expressions

Looking at label_expression in Fig. 4 it is possible to obtain nested label expressions, i.e. within a label expression there is another label expression surrounded by parentheses. First, let us show what extra expressive power that gives us. Let example query Q_1 be as follows:

```
MATCH (a:label1 | label2 & label3 | label4)
RETURN *
```

and let example query Q_2 be as follows:

```
MATCH (a:label1 | label2 & (label3 | label4))
RETURN *
```

Now, Q_1 and Q_2 will have different label sets in their node pattern as in Q_1 operator & has precedence over the operator $|$, whilst in Q_2 the most inner label expressions has precedence over &. Therefore, the node pattern for Q_1 has the label set $L_1 = \{\{label1\}, \{label2, label3\}, \{label4\}\}$ and the node pattern for Q_2 has the label set $L_2 = \{\{label1\}, \{label2, label3\}, \{label2, label4\}\}$. Clearly, $L_1 \neq L_2$, however, less obvious is maybe the construction of L_2. Let me compare the $|$ and & operator to their set equivalents resp. $\cup$ and $\cap$. Now, it becomes more obvious, as we can just use the distributive law for sets. Now let Q_3 be the distributed equivalent of Q_2:

```
MATCH (a:label1 | label2 & label3 | label2 & label4)
RETURN *
```

Hence, it is always possible to rewrite a query with a nested label expression to its distributed equivalent, so the interpreter does not allow for nested label expressions without losing any expressive power.

A.3 Where clause in element predicate

The last part of the defined syntax that is rewritten is the where clause in elt_predicate defined in Fig. 2. Such a where clause is always contained inside elt_pattern_filler containing a variable name, hence, in a node or edge pattern that is bound to a variable name. Additionally, it may only make a reference to the variable name of the node or edge pattern it is contained in. Therefore, it is always possible to extract the where clause to a where clause in the match clause. Let me illustrate the first case with the following example query Q_1:

```
MATCH (a WHERE a.name = "Olof")
RETURN *
```

Now, Q_1 can be easily be rewritten to Q_2:

```
MATCH (a)
WHERE a.name = "Olof"
RETURN *
```

B Class diagram of interpreter

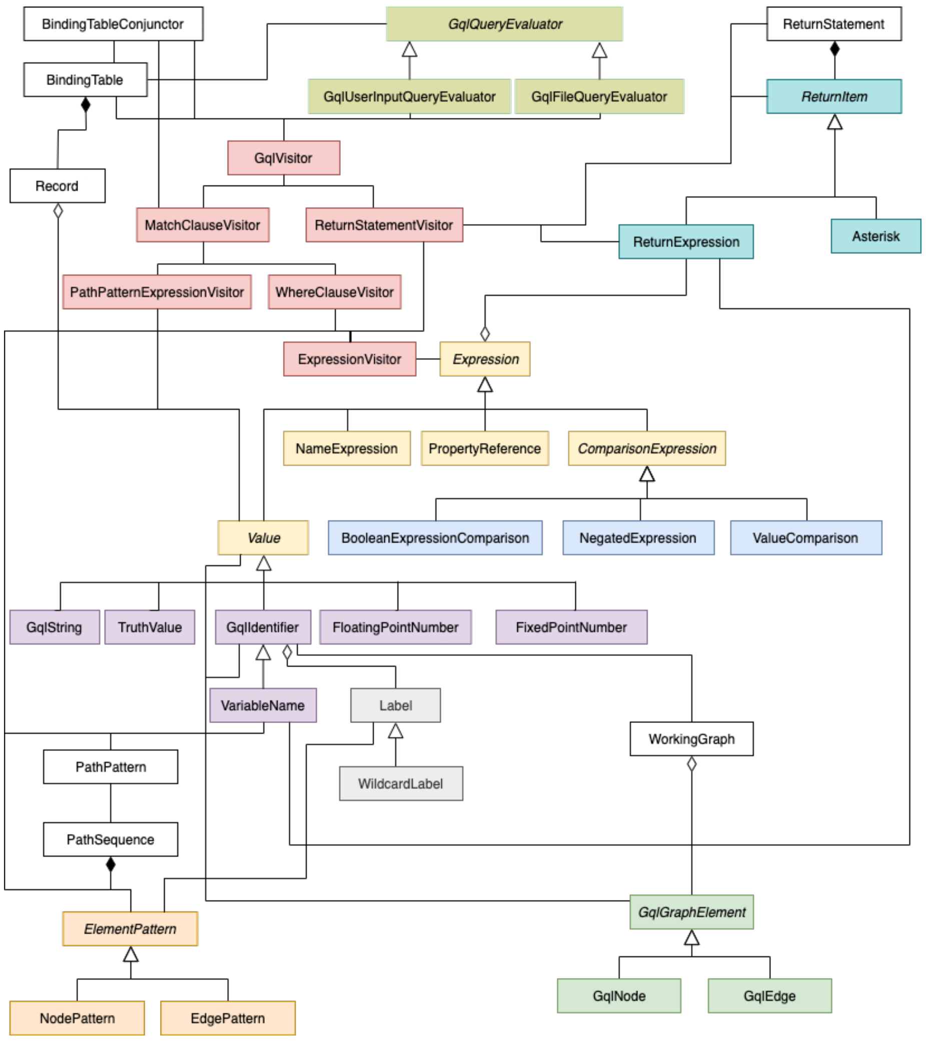


Figure 11: Class diagram interpreter

Above I present a rough class diagram of the implementation of the interpreter [5]. First, a query is obtained and then evaluated by one of the *GqlQueryEvaluators* in kakigreen. From there a parse tree is constructed that is walked by the GqlVisitor. The red classes are all based on the base visitor generated by ANTLR and walk over a syntax tree like the one presented in Fig. 9. Moreover, each visitor constructs objects that represent elements of the semantics.

Specifically, *Value* represents value semantics and NameExpression represents name semantics defined in Section 4.1.4. PropertyReference represents the semantics of maps defined in Section 4.1.2. The semantics of Section 4.1.3 and Section 4.1.4 nulls and truth values is represented by BooleanExpression-Comparison and NegatedExpression. Whereas the comparison of base types and identifiers in Section 4.1.4 is represented by ValueComparison.

Additionally, each base type is represented by a purple class, therefore, the underlying implementation of each type is abstracted from the implementation of Java. For example, a GqlString is compared via an implementation of the Unicode Collocation Algorithm [7] and not as Java String. Similarly, labels have their own implementation. Also, the return semantics in Section 4.2.1 is represented by ReturnStatement.

Clearly, Definition 2.1 and 2.2 are represented by their own classes and a path pattern is also represented by its own class. The match semantics from Definition 2.3 is implemented in these classes as well.

The evaluation result is of course stored in a binding table that follows Definition 1.2. The semantics of query conjunctions from Section 4.2.2, Section 4.2.3, Section 4.2.4 and Section 4.2.5 are defined in BindingTableConjunctor. That class also includes the implementation of Definition 2.4 as that definition is the cross product of two binding tables.

At last, the WorkingGraph class is a Singleton class as there is only one working graph that is set to a graph from the database during runtime. Besides, using the Singleton design pattern allows the working graph to be available from all classes and be the same instance at all times. The working graph has a name and consists of nodes and edges that use the same classes as the semantic representations. This way the evaluation of the semantics becomes trivial.

C Graph database format

A query interpreter cannot exist without data to evaluate on. As the interpreter is written from scratch and the evaluation is done in Java, the data must also be parsed into Java. Therefore, I have created a custom JSON parser for node and edge files. In this section I will elaborate on the format of the current database.

As the aim of this project is not speed, but exploration and consistency along the GQL ISO Standard [21] the database is not optimized for large datasets. In the GitHub repository [5] one can find the database folder under *src/main/resources*. Each folder inside the database folder is a graph where the folder name should be a valid GQL identifier as that is used as graph name in the interpreter. For example, folder *g3* can be referenced from a GQL query using the statement:

```
FROM g3
```

Each folder in the database must contain two files, one containing the nodes named *nodes.json*, and one containing the edges named *edges.json*. Both JSON files have a similar structure, but are not exactly the same. Let me elaborate on both separately.

C.1 nodes.json

A node is an object n that must contain a field *identifier* containing its id within double quotes, and optionally a field *labels* that has a list in square brackets containing labels within double quotes, and optionally a field *properties* that is a map with key-value pairs where the key is an id in double quotes and the value a truth value inside double quotes, word or number that is read exactly the same as in a GQL query. In folder *src/test/resources/database/fullExampleGraph* in the GitHub repository [5] one can find the most extensive example.

C.2 edges.json

An edge is an object e that has the same options as a node with some additions. It also must contain a field *startNode* that contains a node id within double quotes that is present in the corresponding *nodes.json* file. Similarly, it must contain a field *endNode*. At last, it may contain a field *isDirected* that may contain a 0 (undirected) or 1 (directed). In folder *src/test/resources/database/-fullExampleGraph* in the GitHub repository [5] one can find the most extensive example.

C.3 Graph selection

Obviously, queries need to be evaluated on a graph. Moreover, in GQL a query is evaluated on the current working graph, a single graph that is selected by the system or user. Queries can be focused, i.e. contain a `FROM` clause, or ambient, i.e. not contain a `FROM` clause. A focused query changes the working graph to the selected graph in the **FROM** clause, where an ambient query directly evaluates on the current working graph. The first selection of the working graph is done by the system, it is always the graph in *src/main/resources/database* folder that comes first after sorting the folders in ascending alphabetic order.

Now, let me exemplify the above by an example query $Q1$ that is a union of query $Q1_a$ and $Q1_b$:

```
MATCH (x)
RETURN *
UNION
FROM g3
MATCH (x)
RETURN *
```

At the start of executing this query in the console using the interpreter, the first working graph that is selected is g. Hence, $Q1_a$ is evaluated on g. On the other hand, $Q1_b$ first changes the working graph to $g3$ and then evaluate the query. Now, the working graph is $g3$ until the interpreter is shut down in the console or the working graph is changed by the user using a **FROM** clause.