

# Linked Data Council's Business Intelligence and Interactive Benchmark from GSQL to GQL

Ryan Meghoe  
email <mailto:r.a.meghoe@student.tue.nl>

August 27, 2022

Supervisors:  
Dr. MingXi (TigerGraph)  
Professor George Fletcher (TU/e Database Group)

# 1 Introduction

## 1.1 Graphs

Representation of sophisticated network structured data is possible through graphs. Graphs are used in social networks, biology [DFG<sup>+</sup>22], and chemistry[ABD<sup>+</sup>21]. Moreover, graphs are seemingly an interesting way to model data to many users, because of the similarity in the way humans model data. The representation of graphs can be done through two well know frameworks; RDF [W3C22a] or Property-Graphs [BFVY18]. In the context of enterprise data management, many current graph database systems offer support property graphs, such Neptune, Neo4j, TigerGraph, Oracle Server[DFG<sup>+</sup>22]. A property graph is a multigraph where nodes and edges have labels and properties.

## 1.2 Motive

Since many database vendors are involved in the processes, standardization is crucial and started in 2019. The development of standards for property graphs is in process. ISO/IEC JTC1 approved a project to standardize a property graph database language named Graph Query Language(GQL). Aside from standardizing the language, the data model, schema and constraints also need to be taken into consideration. At this point, the focus is to standardize a graph query language for property graph schema and constraints. Since operability is of major importance between various graph technologies, the risks must be minimized. Vendors are implementing their iterations of schema and constraints as a result of the rising industry adoption of graph databases. The dispersion will be so great when the ISO committee begins discussing schema and restrictions that it will be challenging to align disparate methods. We recognize the need for standardized property graphs, schema, and restrictions that offer full CRUD options as a community of graph database industry practitioners and academics.

## 1.3 Benchmark for Graphs Databases

GQL is composed of other Graph Languages, such as a part of SQL named SQL/PGQ [DFG<sup>+</sup>22]. Due to this, a new feature is added to the existing version of GQL, namely allowing a select statement, and will be further discussed in 5. GQL and SQL/PGQ share the same data model, and graph pattern matching language (GPML). GPML is a graph pattern language in which the main principle of operation is on path bindings. The projection of path bindings between GPML, GQL, and SQL/PGQ differs. However, the path pattern for a graph remains similar for all three languages. The whole standardization process is governed by the WG3, which has an association with the Linked Database Council Community(LDBC). The LDBC is a network of industrial companies, academic researchers, and consultants, that mainly design benchmarks for graph data workloads. Graph workloads aimed at database management systems are defined by the Social Network Benchmark suite. Since there are two separate workloads, the benchmark suite consists of two unique benchmarks operating on a single dataset. The Social Network Benchmark's Interactive workload is the first one and focuses on transactional graph processing with sophisticated read queries that examine the node's surrounding area. Furthermore, it updates operations that repeatedly add new data to the graph. The Business Intelligence query for the Social Network Benchmark concentrates on complicated aggregation- and join-heavy queries that touch a significant percentage of the graph in micro-batches of insert/delete operations.

## 1.4 Scope and Overview

In [Mor21]the formal semantics is defined for GQL, and a naive GQL parser is built. In this project, the previously realized project will be referenced. In comparison to his work, this project focuses more on syntax level, and pattern matching, rather than a deep dive into the semantics. This project will focus on the LDBC Social Network Benchmark, and its main objective is to rewrite the 46 [GSQ22a] [GSQ22b] queries to GQL, written in GSQL. GSQL is the database language of TigerGraph. Unlike other graph database technologies, TigerGraph offers aggregation support through containers called Accumulators[DXWL20]. A dive into accumulators, their descriptions, and the principle of operation is provided in 5.

Accordingly, this report will be structured as follow: Chapter 2 describes the data model using a brief description of the semantics. Subsequently, in chapter 3 the graph pattern matching is explained with the related syntax. Followed by path bindings in GPML, in Section 4. Section 5 describes the conversion from GSQL to GQL, syntax, and accumulators. Chapter 6, tests the validity of the queries and and describes the limitations and workaround for it. At last, future work, discussion and conclusion are presented in chapter 7.

## 2 Data Model Graph Databases

The important elements of GQL are the values, graphs, tables and views. The query language is also of importance to GQL, which consist out of patterns, expressions, clauses and queries. GQL is to a certain degree similar to the Cypher query language and queries data from property graphs 2.2. The output of an executed query in GQL is a set of path bindings. From the computed path binding a reduction follows and afterwards a table is returned to the user. In order to realise this GQL uses output of the GPML processor to produce the final output. Aside from returning a table, GQL can also return a graph view or a new graph, as depicted in figure 1

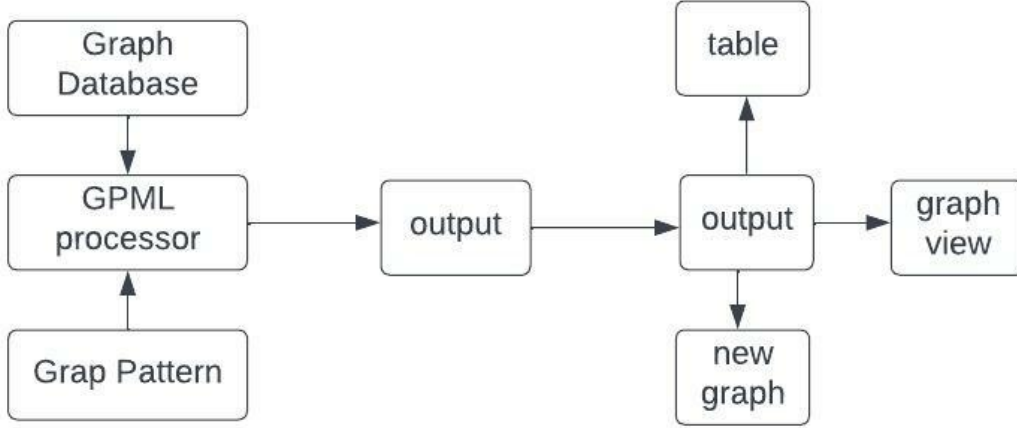


Figure 1: Conceptual Diagram of GQL and GPML

### 2.1 GQL Values

The values of GQL are closely related to Cypher's model, while the data types are similar to SQL. In an environment related to GQL, there are multiple sets, such as property keys, nodes, edges, and a set of names. Moreover, there is the direction that uses 1 to indicate directed, while 0 stands for undirected. The set of values ( $V$ ) in the current version of GQL.

1. Base types(real numbers, finite strings, and floating point numbers ) are considered as values.
2. Node and Edge identifiers are considered values
3. If a  $\text{set}()$  is not empty, and the values contained within that set are distinct, then the  $\text{set}(v_m, \dots, v_m)$  is a value
4. In contrast to a multiset, the same condition applies with distinct values excluded from the conditions.
5. A  $\text{map}()$  is considered a value if  $k_1, \dots, k_m$  are distinct.
6. A node identifier  $n$  with a  $\text{path}(n)$ ; is a value if  $n$  is a node identifier.
7. If the set of the node identifiers and edge identifiers is not empty and there exists a source node  $n_i$  and target node  $n_{i+1}$  then the  $\text{path}(n_1, e_1, n_2, \dots, n_{m1}, e_{m1}, n_m)$  is a value.

For the concatenation of paths, "." is used, and as a condition that the second path must continue from the first path.

## 2.2 Property Graph

In GQL all graphs are property graphs, but for brevity, the term graph is used. There are numerous ways property graphs are described and defined by various sources. In [DFG<sup>+</sup>22] the property graph is described by comparison to a graph (in graph theory) that consists of a set of vertices and nodes. The  $V$  and  $E$  are either two-element subsets when it comes to undirected graphs, while for directed graphs, they can be a pair of vertices.

A property graph is in simple terms defined as a multi-graph, in which multiple edges are possible between two nodes and their endpoint [?]. Also, they are described as pseudo graphs, in which there is an edge looping, between a node and itself. Furthermore, [DFG<sup>+</sup>22] describes a property informally as a graph that is mixed, or it is partially directed. The edge present in the graph can either be directed or undirected, or source and target nodes can be present. If such nodes are present in the graph it becomes a directed version. At last, another characteristic of a property graph described is that the nodes, edges, and labels can have attributes. The formal definition for a property is derived from [Ang18] and goes as follows: Assume that  $P$  is an infinite set of property names,  $V$  is an infinite set of atomic values,  $T$  is a finite number of data types, and  $L$  is an infinite set of labels (for nodes and edges) (e.g., integer). We suppose that  $SET+(X)$ , given a set  $X$ , contains all finite subsets of  $X$  aside from the empty set. The function  $type(v)$  returns the data type of  $v$  for a value  $v \in V$ . The quoted strings are used to identify the values in  $V$ .

Definition 1.1 [GGL21]Property Graph A property graph is defined as tuple  $G = (N, E, rho, \lambda, \sigma)$  and :

1.  $N$  is a finite set of vertices

2.  $E$  is a finite set of edges

3.  $\rho : E \rightarrow (N \times N)$

$\rho$  represents a function that combines each edge in  $E$  with a pair of vertices in  $N$

4.  $\lambda : (N \cup E) \rightarrow SET+(L)$ , where  $\lambda$  is a labeling function that combines a vertices or edge with a set of labels from  $L$ .

5.  $\sigma : (N \cup E) \times P \rightarrow SET+(V)$

$\sigma$  is a function that combines nodes or edges with properties, and for each property it assigns a set of values from  $V$ .

Bonafeti Et al further states that nodes and edges are pairwise independent.

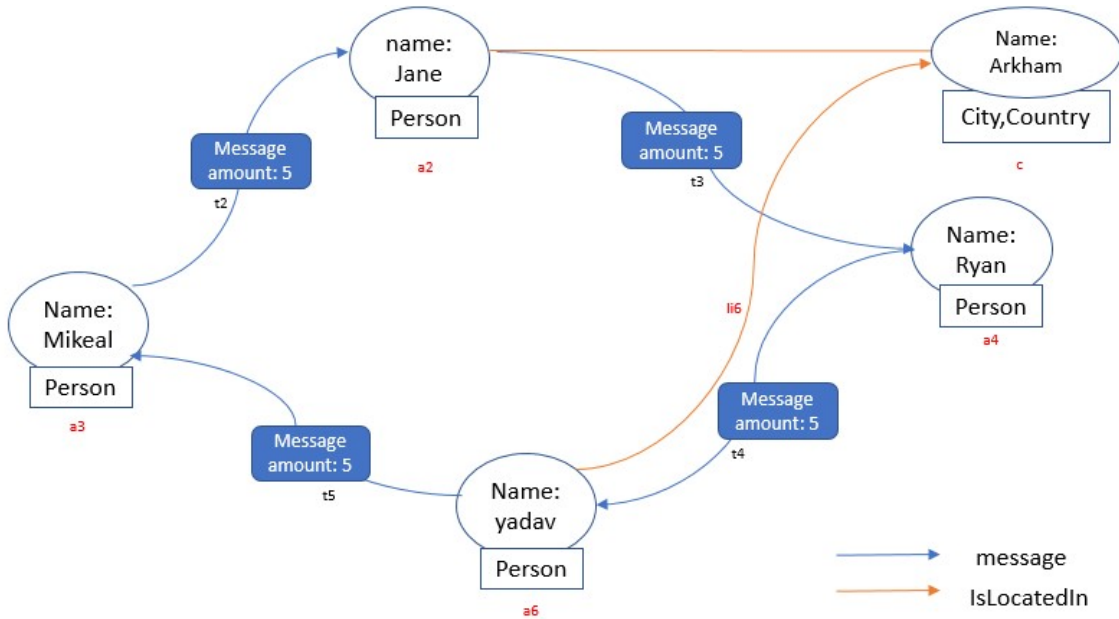


Figure 2: Propertygraph

## 2.3 Binding Tables

GQL also supports binding tables, like Cypher and SQL. One important aspect of binding tables is that after construction, such are immutable [Har21]. The author of [Mor21] implemented the functionality of binding tables, which are returned as a result of the GQL queries, rather than row results. A binding table is a table structure where the body of the table provides bindings for the variables in the head of the table [Har21].

As Mora defines a record as an incomplete function from names to values represented as a tuple with named fields. The name fields are  $r = (a_1 : v_1, \dots, a_n : v_n)$ . Out of the name fields we have  $a_1, \dots, a_n \in A_2$ , which are distinct names, and  $v_1, \dots, v_n \in V$  are values.

In his work, the author implements a similar concept. Each name is defined as  $a_i$ , where  $i$  denotes the set of pairs of names. The pair of names are noted as  $a_1 \cdot a_2$  rather than  $a_1, a_2$ . All the names,  $a_1$  do refer to a graph name in a schema  $S$ , while an edge or node pattern is mentioned by  $a_2$ . The domain of a tuple  $r$  is defined and referred to as the set of names  $a_1, \dots, a_n$ . The unit value is a record  $r = ()$  with zero fields. If an execution is done on a graph  $D(s)$ , an abbreviation is applied that leads from  $s \cdot a_i$  to  $a_i$ . Moreover, there is a function  $\delta: T \rightarrow T$ , that maps a table to itself, where all duplicates are eliminated. However the current version of GQL does not use binding tables to persist data, but rather as a primary iteration construct used for the execution of the procedure. Moreover, binding tables are viewed in GQL terms as a placeholder for intermediary results, that the match clause produces through its pattern matching, and returns the results.

If we have three match clauses on figure 2, which are :

- MATCH (p1:Person)-[:isLocatedIn WHERE city= 'Arkham']-(p2:City)
- MATCH (p1:Person)-[:message]-(p2:Person)

The above match pattern result in :

- T1 = ( $\{p1, p2\}, \{p1:a6, p2:a2\}$ )
- T2 = ( $\{p1, p2\}, \{p1:Ryan, p2:Yadav\}$ )

The union of T1, T2 which is T3 becomes (as multiset):

T4 = ( $\{p1, p2\}, \{p1:a6, p2:a2\}, \{p1:Ryan, p2:Yadav\}$ )

## 3 Graph Pattern Matching

This section describes and discusses the syntax of pattern matching, rigid patterns, path patterns, path construction, quantifier and group variables, conditional variables, and restrictors and selectors.

### 3.1 Pattern Matching

[DFG<sup>+</sup>22] uses the acronym GPML to describe the pattern matching language. The GPML language which stands for graph pattern matching language goes as follows:

**MATCH** pattern

In *pattern* the specification is done on the graph and its components that we want to query. Patterns can be node or edge patterns. For node patterns the definition goes as follows:

**Definition 1.**  $[?]$   $a$  is a node pattern denoted as  $\chi$  and is a triple of  $(a, L, P)$  and :

- $a \in A \cup \{nil\}$  is an optional name.
- $L \subset \emptyset \cup \{L_1, \dots, L_n\}$ , which is finite
- $I$  can be  $nil$  or  $(m, n)$  where  $m, n \in \mathbb{N} \cup \{nil\}$

The node patterns allow users to retrieve a specific node from a graph.

### MATCH (X)

The variable inside the brackets is called an element variable. Through that variable bindings can be formed from the variable as a starting point to another node property in the graph or labeled node. For specification of label nodes, the name of the labels follows as :

**MATCH** (node-name: label-name)

In addition for labels within the node pattern a WHERE clause is possible, which is used to filter on specific label properties. The place of the WHERE clause does matter. If it is placed within the MATCH clause then it acts as a pre-filter, while outside the MATCH clause it acts as a post-filter.

**MATCH**(person: Person)  
**WHERE** person.birth-date > 2000

Here the person predicate filters the results after they are retrieved from the person node.

**MATCH**(person: Person) **WHERE** person.Birthdate>2000)

Here the filter is carried out during the search.

The empty match clause in itself without any node specification is pointless. But in the combination of edge patterns, relationship patterns can be constructed and extracted. Those make it possible to form a connection between two nodes and are defined as :

**Definition 2.** [FGG<sup>+</sup> 18]***Relationship Pattern** An relationship pattern  $\rho$  is an tuple that consist out of  $(d,a,L,P,I)$  where:*

- $d \in \{\rightarrow, \rightarrow, -\}$  indicate the direction of the edge. For  $-$  is used to denote the undirected direction, while  $\leftarrow$  the left direction, and  $\rightarrow$  the right direction.
- $a \in A \cup \{nil\}$  represents a name; which is optional.
- $L \subset \emptyset \cup \{L_1, \dots, L_n\}$  for which each node label can be a finite empty finite set.
- $P$  can be an empty finite set of key-value pairs, in the form  $(k,v)$ ;  $k, \in \kappa$ ,  $v, \in \nu$
- $I$  can be nil or  $(m,n)$  where  $m,n \in \mathbb{N} \cup \{nil\}$

The connection of two nodes formed by an edge pattern is done as denoted as :

**MATCH** -[edge-name : label-name]->

The pattern looks for all directed edges in the graph and binds them to the element named edge-name. Also in the edge pattern, the **WHERE** clause can be used either in or outside of the pattern.

Edge Pattern	Orientation
<-[specification]-	left directed
<~[specification]~	left undirected
-[specification]->	right directed
~[specification]~>	right undirected
~-[specification]~	undirected unspecified left or righth

Table 1: Types of Edge Patterns

### 3.2 Path Pattern Union and Multiset Alternation

The GPML supports two forms of unions. The first union form is known as path patterns union, and the second is the multiset alternation. In the path pattern union the vertical bar is used, while for the multiset the  $|+|$ . The main difference between those two is the path binding formed. In the property graph constructed in 2 there are two city nodes present

The query as shown below with the infix vertical bar produce three path bindings, that in the end will be deduplicated or reduced to two path bindings

MATCH (country: Country)| (city:City)

The first path binding that is formed is from  $(c \rightarrow \text{country1})$ , and the second is  $(c \rightarrow \text{country2})$ . A third path, which is not a duplicate, the path  $(c \rightarrow \text{city})$  is formed. Contrary to the path pattern union, multiset alternation does not carry out deduplication. If the MATCH pattern above is rewritten with a multiset alternation, then there will be three path bindings. The path pattern union looks similar to the condition variable  $|$ . However, it is not the same, and in order to use it must be defined in combination with square brackets  $[ ]$ . By using the  $|$  we can create singletons such as the conditional and unconditional singleton.

MATCH [(node1)->(node2)] | [(node1)-> (node3)]

Node1 binds when one of the path pattern union binds. Nodes 3 and 2 are not bounded by each other, and only by node 1. In this scenario, we say that node 1 is an unconditional singleton while the other two nodes are conditional singletons.

Bear in mind that the usage of implicit equijoin is prohibited, such as

MATCH [(node1)->(node2)] | [(node1)-> (node3)], (node1)->(node4)

To introduce a conditional singleton the  $?$  is utilized as a postfilter MATCH (node1) [->(y)]? The  $?$  operator is almost equivalent as the 0,1 quantifier. However the main difference lies in the exposure of the variables. Singletons are exposed as conditional variables, while the quantifier as group variables.

### 3.3 Restrictors and Selectors

In order to prevent infinite matches restrictors are introduced as path predicates. Restrictors can be constructed at the start of parenthesized path pattern or at the start of a path pattern as shown in the following examples:

**MATCH** Type-Restrictor  
...remaining query

**MATCH** Type-Restrictor  
a = (person1 WHERE person.id = '20329') <-[:KNOWS]-[person where  
person1<>person2]

Restrictor	Morphism	Description
TRAIL	Edge	repetition of edges not allowed
ACYCLIC	Node	repetition of nodes not allowed
SIMPLE	Partial	only the first and last node can be the same, further no repeated nodes

Table 2: The morphism is described as the type of morphism between the graph  $G$  and the path pattern  $\pi$

Contrary to a restrictor, selectors partition the obtained results and from it a finite set is selected. Moreover the selector can be specified before the WHERE clause or after. If the selector is applied before a selection then it serves as a pre-filter. Utilizing a selector after the final WHERE clause it acts as a post-filter.

The major difference between the two concepts is that [DFG<sup>+</sup>22] describes restrictors as the operation that is executed during the pattern matching, and selectors are carried out after. Both can be used in combination under the condition that the restrictors are specified after an selector.



Keyword	Description
<b>ANY</b>	One path is selected randomly in each partition
<b>ANY K</b>	random k path are selected in each partition. If the number of paths < K then all the paths are selected
<b>ANY SHORTEST</b>	One path is selected from each partition that has the shortest length
<b>ALL SHORTEST</b>	In each partition the shortest path is selected
<b>SHORTEST K</b>	The shortest K paths are selected
<b>SHORTEST K GROUP</b>	Each partition is based on the target point, and paths of same length are grouped and sorted.

Table 3: Selectors and their description

**MATCH** Type-selector Type-restrictor  
.....remaining query..

Another use of restrictors and selectors is to ensure the termination of a GPML. Moreover there is a strict rule that every unbounded variable \* must be used in combination with a restrictor or selector.

MATCH r= (person WHERE person.birthdate= 1995-[:KNOWS]-> \*

The query above will result in infinite results since it will include any length of path traversed through the labeled node.

To ensure path termination the current version of GQL will not allow all predicates on unbounded groups.

### 3.4 Quantifiers

On a singleton or parenthesized path pattern the use of quantifiers is possible. Node and edge patterns are not strictly alternated in the pattern by a quantifier. In order to truly notice its influence it is suggested by [DFG<sup>+</sup>22] to leave the source and target node unspecified, and do as much of the filtering needed in the label specification area as shown below:

MATCH (p:Person) [()-[wiring:Transfer]-> **WHERE** count(Comment)>5] {3,4}

The unspecified nodes are also called anonymous nodes. Furthermore, the variables in the edge patterns are referred to as singletons. In the above code example, the variable wiring remains within the edge pattern. However, if it is used past the quantifier {3,4} it is referred to as a group reference.

MATCH (p:Person) [()-[comment:Comment]-> **WHERE** comment.count(Comment)>5] {3,4}  
(p1:Person) WHERE AVG(comment)> 4.5

Notice that the comment variable passes the quantifier and hence is referenced as a group reference.

{m,n}	between m and n repetitions
{m,}	m or more repetitions
*	same {0,}
+	same as {1,}

Table 4: Quantifiers, {m,n } ∈ ℕ

```

1 MATCH TRAIL
2 WHERE p.person = 'Ryan'
3 [-[m: Message WHERE count(m.message) > ]-> 20]+
4 (p) [-[:isLocatedIn]->(place:City) |
5      -[:isLocatedIn]->(c:Country)]

```

Listing 1: Code example retrieved from the property graph in 2

## 4 Path Bindings

Path bindings are a series of elementary bindings. These bindings are a pair of variable and graph elements. The result(s) from a query in GQL is a (multi)set or reduced path bindings. The necessary steps to determine path bindings are normalization, expansion, rigid pattern matching, reduction, and de-duplication. In the current sections, the following query will be used and modified to explain those sections.

To guarantee that the query terminates the TRAIL quantifier is used and looks for sequences of messages of any length, that start and end with a person named Ryan. The variability in path length also counts for city and country nodes.

### 4.1 Normalization

It is crucial in the normalization step that the pattern between node and edges are consistent, such that each sequence has to start with a node pattern and end with it, or an alternation between a node and edge patterns is possible. GPML makes it easier to aid in the construction of patterns by noting them into a canonical form. Moreover, the  $+$  is replaced by the quantifier  $\{1,0\}$ . Thus, the pattern is rewritten to:

$$\begin{aligned}
 & \text{MATCH TRAIL ( p WHERE p.person = 'Ryan')} \\
 & ()-[m: \text{Message WHERE count(m.message)->20 }]-> () \{1,0\} \\
 & (p) [ ()-[:\text{isLocatedIn}]->(\text{place:City}) | \\
 & \quad ()-[:\text{isLocatedIn}]->(\text{c:Country})]
 \end{aligned}$$

Do note that there are three empty nodes(anonymous) added compared to the initial query in 1. Now the pattern is rewritten by adding a new variable with an index,  $\square_x$ . The pattern rewritten becomes :

$$\begin{aligned}
 & ( p \text{ WHERE p.person = 'Ryan'}) \\
 & [(\square_i)-[m: \text{Message WHERE count(m.message)->20 }]-> (\square_{ii}) \{1,0\} \\
 & (p) [(\square_{iii})-[:\text{isLocatedIn}]->(\text{place:City}) | \\
 & \quad (\square_{iv})-[:\text{isLocatedIn}]->(\text{c:Country})]
 \end{aligned}$$

### 4.2 Expansion

An extension of the pattern happens by a set of rigid patterns, that does not contain any form of disjunction. It makes sense that a rigid pattern would be one that a SQL equijoin query could convey. Officially it is a pattern without quantifiers, union, or multiset alternation. Additionally, the expansion annotates each rigid pattern to make it possible to follow the origin of the syntax constructions. The number of iterations for each quantifier is adjusted through rigid patterns. Moreover, the number of iterations is modified for alternations and disjuncts of unions. In the initial code 4 the disjunct of the union is the  $|$ , and the quantifier  $1,0$  is expanded. To prevent the query from providing many endless possibilities, it might be a viable solution. An option to prevent this query from looping or giving infinite possibilities is to modify the left side of the dis-junction  $()$ .

$$\begin{aligned}
& (p \textbf{ WHERE } p.\text{person} = \text{'Ryan'}) \\
& (\Box_i^1) - [m^1 : \text{Message} \textbf{ WHERE } \text{count}(m.\text{message}) > 20] \rightarrow (\Box_{ii}^1) (p) \\
& (\Box_{iii}) - [-_i:\text{isLocatedIn}] \rightarrow (\text{place:City})
\end{aligned}$$

The quantifier is expanded, such that  $n \in N \setminus 0$  and for each path pattern there is either chosen from the city or country node of the path pattern union. The notation of the pattern is denoted as  $\theta_{n,city}$  or  $\theta_{n,country}$

$$\begin{aligned}
& (p \textbf{ WHERE } p.\text{person} = \text{'Ryan'}) \\
& (\Box_i^1) - [m^1 : \text{Message} \textbf{ WHERE } \text{count}(m^1.\text{message}) > 20] \rightarrow (\Box_{ii}^1) \\
& (\Box_i^2) - [m^2 : \text{Message} \textbf{ WHERE } \text{count}(m^2.\text{message}) > 20] \rightarrow (\Box_{ii}^2) \\
& (\Box_i^n) - [m^n : \text{Message} \textbf{ WHERE } \text{count}(m^n.\text{message}) > 20] \rightarrow (\Box_{ii}^n) \\
& (p) \\
& (\Box_{iii}) - [-_i:\text{isLocatedIn}] \rightarrow (\text{place:City})
\end{aligned}$$

In each iteration a superscript is present, and shortly after a cleanup follows. The symbol reserved for the pattern is  $\pi_{n,l}$  where  $l \in \{City, Country\}$ . In order to have a neat cleaned up query, each anonymous node pattern that is adjacent to another anonymous node pattern is removed.

$$\begin{aligned}
& (p \textbf{ WHERE } p.\text{person} = \text{'Ryan'}) \\
& - [m^1 : \text{Message} \textbf{ WHERE } \text{count}(m^1.\text{message}) > 20] \rightarrow (\Box_{ii}^1) \\
& - [m^2 : \text{Message} \textbf{ WHERE } \text{count}(m^2.\text{message}) > 20] \rightarrow (\Box_{ii}^2) \\
& - [m^{n-1} : \text{Message} \textbf{ WHERE } \text{count}(m^{n-1}.\text{message}) > 20] \rightarrow (\Box_{ii}^{n-1}) \\
& - [m^n : \text{Message} \textbf{ WHERE } \text{count}(m^n.\text{message}) > 20] \rightarrow (\Box_{ii}^n) \\
& (p) \\
& - [-_i:\text{isLocatedIn}] \rightarrow (\text{place:City})
\end{aligned}$$

Notice how the anonymous node patterns on the left side of each line in the code above is removed as a cleanup step.

### 4.3 Rigid Pattern Matching

Path bindings are computed for each rigid pattern. Through variables of the same name by a join, computation is executed by taking the elementary construct of the rigid pattern. The definition of path bindings is given in 4. Elementary bindings, which are another name for path bindings are a pair of variables and graph elements. Those elements are portrayed as tables. In the first row of the table are variables, and in the second the graph elements are denoted, for example:

$$\begin{array}{ccc}
a & b^1 & \Box_{ii}^1 \\
a4 & t4 & a6
\end{array}$$

During the operation for evaluation of rigid pattern the node-edge-node relationship below is evaluated and delivers as result the path binding given above.

$$\begin{aligned}
& (p \textbf{ WHERE } p.\text{name} = \text{'Ryan'}) \\
& - [b^1 : \text{Transfer} \textbf{ WHERE } b^1.\text{count}(\text{message}) > 20] \rightarrow (\Box_{ii}^1)
\end{aligned}$$

Other independent path bindings computed that belong to  $\pi_{4,city}$ . Since the In the evaluation process, each label is checked and the where condition. From an evaluation of  $\pi_{4,city}$  the following path bindings arises:

The above path binding is filtered based on the WHERE and the label constraint. Eventually variables with the same name are joined based on a implicit equi-join. The equijoin becomes :

$\square_{ii}^1 b^2 \square_{ii}^2$	$\square_{ii}^1 b^2 \square_{ii}^2$	$\square_{ii}^3 b^4 \mathbf{a}$	$\mathbf{a} -_i \mathbf{c}$
a6 t5 a3	a6 t5 a3	a6 t5 a3	a4 li4 c2
a3 t2 a2	a3 t2 a2	a3 t2 a2	a6 li4 c2
a2 t3 a4	a2 t3 a4	a2 t3 a2	a3 li3 c1

(4 more)      (4 more)      (4 more)      (3 more)

$\mathbf{a}$	$b^1$	$\square_{ii}^1 b^2$	$\square_{ii}^2 b^3$	$\square_{ii}^3 b^4$	$\mathbf{a}$	$-_i$	$\mathbf{c}$
a4	t4	a6	t5	a3	t2	a2	t3
							a4 li4 c2

## 4.4 Reduction and deduplication

In the final step in retrieving the results of a GQL, query reduction or deduplication is done. This is done by looking at the path bindings that are matched by the rigid patterns. The bindings are reduced, by removal of the annotations (removing the super and sub scripts) and are collected into a set. All the element patterns that are anonymous pattern are merged. An important restrictor to apply is the use of a selector after the reduction or deduplication is achieved. A downside of deduplication is that it can combine queries with path pattern union such as the following:

(p)- [:isLocatedIn]->(city:City)- [:isLocatedIn]->(country:Country)

becomes after deduplication :

(p)- [:isLocatedIn]->(City |Country)

In order to prevent this it is advised to use the multiset path alternation

(p)- [:isLocatedIn]->(city:City) **||** - [:isLocatedIn]->(country:Country)

## 5 Graph Query Language

In the work of [Mor21], the full semantics of GQL is defined. Since in this project the scope is mainly about converting and utilizing the GQL parser, an in-depth discussion about the semantics will be left out.

### 5.1 GQL features

A GQL query consists of the MATCH clause, as specified in chapter 3. Aside from the MATCH, there is a FROM and Return clause. The FROM clause is used to indicate or select the graph on which the pattern matching has to be executed, and the return clause returns the results. A brief overview of a GQL query is provided below:

```

query ::= query expr (query conjunction query expr)*
query conjunction ::= set operator | OTHERWISE
query expr ::= focused query expr | ambient query expr
focused query expr ::= (FROM a match clause)+ return statement a ∈ A
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)*

```

The recently published documents at the GQL iso standard provide a more simplified form of GQL as :

```

1 SELECT <Results>
2 FROM <graph>
3 MATCH <path pattern>
4

```

Listing 2: GQL Cypher version

Conceptually the first FROM specification in GQL is not allowed to be omitted. However, if left empty the query will MATCH on the graph of the previous query. Two versions of GQL are proposed, of which the first one is approved and presented in the [DFG<sup>+</sup>22]. The major difference lies in the return statement. Compared to the first one there is no additional information given aside :

```

1 SELECT <Results>
2 FROM <graph>
3 MATCH <path pattern>

```

Listing 3: GQL SQL version

In [Neo18] views, and graph construction are mentioned. However, the recent update state that the addition of views to the GQL is approved. But the structure and how are not yet provided in recent sources. During the initial ballot the proposed view was :

Creation of a view:

```

1 CREATE QUERY <name> [<parameter list>] AS {
2   subquery
3 }

```

Listing 4: GQL view

Users might intend to use views to return tables. But in GQL the views are meant to return graphs, or create subgraphs. In views entities become vertexes and relationships become edges. According to [W3C22b] the CALL is used to enable multiple graph queries and goes as follow:

```

1
2 CALL {
3   FROM <graph Name>
4   MATCH <pattern list>
5   RETURN <result list>
6   UNION
7   FROM <graph Name>
8   MATCH <pattern list 1>
9   RETURN <result list 1>
10 }
11 MATCH (:node) WHERE <condition>
12 RETURN c.name AS name, kind

```

Listing 5: Multigraph Construction

As for the view the construction of syntax is provided by [Neo18] and presented as : According to the

```

1 CREATE QUERY viewNAME($input graphReturned) AS
2 FROM $input
3 MATCH node-edge pattern
4 CONSTRUCT (node1)-[:viewNAME]-(node2)
5

```

Listing 6: GQL view graph returned

[LDB21] the option to implement CALL is still under consensus. However views will be added to GQL, according to [DFG<sup>+</sup>22] and the [Gre22]. But how and final syntax or structure is not yet decided upon. Up to this point, there is no additional source that dives into it. Hence the translation and derivation of that language are taken from [DFG<sup>+</sup>22].

## 5.2 GSQL

In contrast to GQL, GSQL uses an accumulator for aggregation. In GQL the query pattern produces a match table with column names determined by the pattern variables, and each row has a binding of these variables to graph elements, where the binding is consistent with a match of the pattern against the graph. In every situation, the match table is combined using a method that is analogous to a standard SQL GROUP BY clause.

Accumulators are data containers that store an internal value and accept inputs that are aggregated into this internal value with the use of a binary operation. There are two types of accumulators; namely the vertex and global accumulator. The vertex accumulator is attached to vertices, and each vertex stores its own accumulator. Global accumulators are used as a single instance and are applied for using global aggregates. Accumulators are implemented using two assignment operators:

- t = i; assigns the value i to input t
- t+=i; aggregates the input i into the accumulator t

Various types of accumulators :

- SumAccum<Values>: The values are of numeric type and store the sum of the values.
- MaxAccum<Values>: The values are ordered and the maximum values are computed.

- `MinAccum<Values>`: The values are of an ordered type and the minimum value is computed.
- `AvgAccum<Values>`: Values are of numeric type and the average is computed.
- `OrAccum`: the logical conjunction is taken from all the values.
- `MapAccum<K, V>`: In this map accumulator each key can map to value `V`, where `V` on itself can be another accumulator.
- `SetAccum`, `BagAccum`, `ArrayAccum`, and `ListAccum` are all accumulators where the values are placed within a collection of a type, for e.g `T`.

At last, one of the frequently used accumulators in [GSQ22a] and [GSQ22b] is the `HeapAccum`. In that, there is a priority queue implemented, where the values are implemented in tuples.

## 6 Validity Testing

This section briefly explains the absence of current functions and complications and workaround for it. Moreover, there is a subsection that proves the validity of the converted query of GSQL to GQL, by utilizing the naive parser. Besides utilization, there is also some additional information provided on how to use the parser, its limitations, and how to resolve those in [Meg22].

### 6.1 Limitations

The first noticeable complication that arises before the start of the conversion is the lack of Accumulator support. Accumulators as addressed in 5.2. The presence of an accumulator makes it convenient to append results of two match clauses, allows seamless, and easier ways to address filtering. In contrast to GQL well-known WITH clause that is proposed in the [Neo18] concept(2017) is still under discussion according to the LDBC [LDB21]. Fortunately, the WHERE clause does offer a limited functionality as an alternative. Another option to append queries is to use the union in combination with another path pattern declared after it. Furthermore, TigerGraph supports function **ISEMPTY** for which further discussion is needed on whether it should be incorporated or not. Another function that needs to be addressed is the **All()**, and **Any()** from Cypher. In SQL the equivalent function is the **IN**. The current version of GSSQL offers no direct function nor implementation, and the workaround is to filter an accumulator. For GQL most likely the IN predicate will be made available to filter out lists, but also requires further consensus. In addition, the creation of views is mentioned in [Neo18], and according to the recent updates of the GQL standards the support of views is approved, but no further supporting document sheds light on the syntax. Hence views will be implemented as described in [Neo18]. Those views will not be used in conjunction with other views yet and are advised to the available union, except and except all functions to obtain the results if possible. The [Neo18] document further supports subgraph creation, and just as for the view, there is no available support that is approved on how to implement. At last, the control flow statements are addressed. Compared to GSQL the standard if-else is not possible and is achieved by case statements. Also for the COALESCENCE function case statements are utilized. There are three ways to use it and are highlighted below:

- 1) For a nullif condition the case is written as:

```
CASE WHEN n1=n2 THEN
      NULL ELSE n1
END
```

- 2) In the presence of COALESCENCE (n1,n2) the workaround becomes:

```
CASE
WHEN NOT n1 IS NULL THEN n1
ELSE n2
END
```

- 3) If there are more than two conditions or variables:

```
CASE WHEN NOT n1 IS NULL THEN n1
      ELSE COALESCE (n2, ..., nn)
END
```

### 6.2 Parser Utilization

The parser developed by Morra Olaf is used in order to test a subset of the translated queries. Due to the current GQL developments phase, cross-validation between Tiger-Graph queries is not possible. The query that is tested, is the Bi-7, which belongs to the Cypher variant. At the moment arithmetic functions and aggregations are not possible with the provided parser. Also, the option to create links between nodes, edges, subgraphs, joining of views, and deletion of tables is not yet implemented Hence, these are left out,



```

1
2 FROM BI
3 MATCH (tag:Tag {name:$tagname} )<-[:HAS_TAG]-(m:Message),
4       (m)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)
5 RETURN rTag.name AS rTag_name,
6        count(*) AS count
7 GROUP BY rTag_name
8 ORDER BY count DESC
9         ,rTag_name
10 LIMIT 100
11 EXCEPT ALL
12 FROM BI
13 MATCH (c1:Comment)~[:HAS_TAG]~(tag1:Tag)
14 RETURN *
```

Listing 7: Full Query

and there might be some minor differences in syntax, between the queries provided at [Mor21] and the ones tested in [Meg22].

In 6.2 the Bi7 query is shown with an except clause. Since the graph construction of the parser is limited to one row of values, the test can be done on one row only, resulting in a higher chance of empty returned results. However, in order to evaluate this query, it is broken down into segments and each is evaluated before the execution of the full query. The first segment is also the first path pattern of the first match clause. In this segment, there is a prefilter construction that returns all the results. Since only one row of properties is available due to the way the graph implementation is developed, the results are either one row or empty. Although this might be a limitation it also makes debugging or testing the validity of the query easier. The results obtained are returned to a binding table. Since both the tag and the message node have the same id (purposely constructed this way), it is expected that the results returned by the query are not empty or none.

```

1 FROM BI
2 MATCH (tag:Tag {name:"Rumi"} )<-[:HAS_TAG]-(m:Message)
3 RETURN *
```

Listing 8: Pre- Except query first path pattern

tag	m
{ "identity": "Tag", "labels": ["Tag"],	{ "identity": "Message", "labels": ["Message"],
"properties": { "id": 14, "name": "Rumi",	"properties": { "length": "4", "browserUsed": "755914244482", "id": 14,
"url": "http://dbpedia.org/resource/Rumi"} }	"creationDate": "2011-11-08T16:05:23.917+00:00", "locationIP": "196.29.42.107",
	"content": "good"} }

Figure 3: Results of 8 formatted

As can be seen in 3 there are indeed results returned, where the ids are matched. Since there was a request to return all the results with the \* (asterisk), indeed all the results are returned. The next phase of the testing is to verify if the second path pattern returns the results accordingly. In contrast to the first

path pattern, this is a more advanced path traversal that executes both left and right path traversal. Also here deliberately the IDs are matched in order to return the results.

```

1 FROM BI
2 MATCH (m:Message)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)
3 RETURN *
```

Listing 9: Pre- Except query second path pattern

m	c	rTag
{ "identity": "Message", "labels": ["Message"], "properties": { "length": "4", "browserUsed": "755914244482", "id": 14, "creationDate": "2011-11-08T16:05:23.917+00:00", "locationIP": "196.29.42.107", "content": "good" }}	{ "identity": "Comment", "labels": ["Comment"], "properties": { "length": "77", "id": 14, "browser": "Firefox", "creationDate": "2012-06-29T23:37:12.826+00:00", "ip": "31.209.179.5", "content": "About Erwin Rommel, one of the mAbouPrince Philip, Duke of Edinburgh, marr" }}	{ "identity": "Tag", "labels": ["Tag"], "properties": { "id": 14, "name": "Rumi", "url": "http://dbpedia.org/resource/Rumi" }}

Figure 4: Results of 9 formatted

The results indeed return the results where the IDs are matched and for the next query there is expected to return an extra column since now the results to be returned are the matching IDs of the first path pattern and the second pattern match.

```

1 FROM BI
2 MATCH (tag:Tag {name:"Rumi"})<-[:HAS_TAG]-(m:Message),
3       (m)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)
4 RETURN *
```

Listing 10: Full pattern list

m	tag	c	rTag
[{"identity": "Message", "labels": [{"Message"}], "properties": {"length": "4", "browserUsed": "755914244482", "id": "14", "creationDate": "2011-11-08T16:05:23.917+00:00", "locationIP": "196.29.42.187", "content": "good"}]}	[{"identity": "Tag", "labels": [{"Tag"}], "properties": {"id": "14", "name": "Rumi", "browser": "Firefox", "url": "http://dbpedia.org/resource/Rumi"}]}	[{"properties": {"length": "77", "id": "14", "creationDate": "2012-06-29T23:37:12.82,6+00:00", "Ip": "31.209.179.5", "content": "About Erwin Rommel, one of the mAbout Prince Philip, Duke of Edinburgh, marr"}]}	[{"labels": [{"Tag"}], "properties": {"id": "14", "name": "Rumi", "url": "http://dbpedia.org/resource/Rumi"}]}

Figure 5: Results of 10 formatted

As can be seen, the results are indeed valid, since each tag returns the id's that are matched and the rtag column is appended to the binding table, also with results matching the id of the other columns. The third step is to test the path pattern after the match clause. The following query is mostly identical compared to the above one, with a difference in the columns requested.

```

1 FROM BI
2 MATCH (tag:Tag {name:"Rumi"} )<-[:HAS_TAG]-(m:Message),
3       (m)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)
4 RETURN rTag.name AS rTag_name

```

Listing 11: Full Pattern List with Column name

Since there was only requested to return the name of the tag node, the name and the corresponding value is returned. This is indeed returned as the result shows.

rTag_name
"Rumi"

Figure 6: Results of 11 formatted

The third step is to test the query with the last step that has the path pattern after the except clause. Since the comment node is a subclass of the message node, the same properties can be expected, with some minor differences in property values.

```
1 MATCH (c:Comment)~[:HAS_TAG]~(tag:Tag)
2 Return *
```

Listing 12: Except ALL Query

c	tag
{ "identity": "Comment", "labels": ["Comment"], "properties": { "length": "77", "id": 14, "browser": "Firefox", "creationDate": "2012-06-29T23:37:12.826+00:00", "Ip": "31.209.179.5", "content": "About Erwin Rommel, one of the mAbout Prince Philip, Duke of Edinburgh, marr"}}	{ "identity": "Tag", "labels": ["Tag"], "properties": { "id": 14, "name": "Rumi", "url": "http://dbpedia.org/resource/Rumi"}}

Figure 7: Results of 10 formatted

As can be seen from the obtained results that this is truly the case. At last, the full query has to be constructed. Do note that in order to use the EXCEPT clause, the results or returned column names must be the same. Hence why the asterisk is substituted by the rtagname.


```
1 MATCH (c:Comment)~[:HAS_TAG]~(tag:Tag)
2 Return tag.name AS rTag_name
```

Listing 13: Except All specific column

+	-----	+
	rTag_name	
+	-----	+
	"Rumi"	
+	-----	+

Figure 8: Results of Listing 13 Bi

For the evaluation of the segment 6.2 the table of 6 and 9 should be examined. Since they both return the same results, the exclusion of two results that are the same should result in an empty table.



+-----+
rTag_name
+-----+
+-----+

Figure 9: Results of Listing 13 Bi

Do note that during the whole evaluation of the BI7, the GROUP BY, ORDER BY and the LIMIT option is left out. This due to the fact that they are not yet supported in the current parser.

### 6.3 Tabular Summarized progress

Below a summary of the queries that were possible to translate is provided in tabular form:

Business Intelligence Workload				Interactive Workload	
	PME	Failed		PME	Failed
1			1		
2			2		
3			3		
4			4		
5			5		
6			6		
7			7		
8			8		
9			9		
10			Insert 1		
11			Insert 2		
12			Insert 3		
13			Insert 4		
14			Insert 5		
15			Insert 6		
16			Insert 7		
17			Insert 8		
18			Short 1		
19			Short 2		
20			Short 3		
			Short 4		
			Short 5		
			Short 6		
			Short 7		
			Short 8		

Table 5: Summarized results. PME stands for pattern match evaluated

## 7 Discussion Conclusion

### 7.1 Discussion and Future Work

**Research Directions** In the recent talk of Alistair [Gre22] views are re-discussed and the extension of what views should return. At the moment views are seen as a function that returns either a binding table of predicate variables or a table, which is exactly what happens in PGQ. However, for GQL this should not be the case and further discussion is required. As proposed GQL should return a path set with binding variables. After that, the graph element can be added or removed. From there on a datagraph can be formed or is returned as a final result of a GQL query, and then views come into play. Due to this, at the current progress of GQL, it is insufficient to translate queries that require views, since there is no explanatory document that further dives into the how.

In the future, speculated in 2024, according to the [W3C22b], GQL views can be used to modify or translate the remaining queries that require such. In addition, those changes can be implemented in the GQL parser. Also, some functions which GSQL uses, or the libraries such as Apoc, limit the possibility to have all the queries evaluated and translated. As for researching direction path patterns should be evaluated at a deeper level, especially in ensuring termination of queries, that use the Kleene star\* and the type of selector. As for now, there is some guarantee that the \* star in combination with the type selector can be an upper bound.

Moreover, since GQL is a new language, TLP which is executed and analyzed on SQL, can be also done for GQL and the PGQ/SQL language in the future when there is a fully complete Database Engine [RS20]

### 7.2 Conclusion

Research on GQL is executed before by Olof Morra. In his project, he aims to develop and address the semantics of GQL. In contrast to his project, this one is a continuation but more on a syntax level and looks at the current development of GQL. Most of the concepts, which were under consensus when he started, are approved. Thus, an additional aim of this project is to convert the 46 queries provided by TigerGraph, which is a leading database vendor and actively participates in the standardization of GQL along with the Linked Database Community Council. To test the validity of the converted queries, a naive database tool will be utilized. Since the tool is limited in many ways, cross-validation between the GSQL language and the GQL language is currently not viable. Hence, the validity will mostly be performed to verify whether or not the queries follow the guideline and are to yield results. To utilize the parser, the user should provide the interpreter with a query, which it will parse according to the specified syntax. A parse tree is created using the syntax generator ANTLR [20], from which a tree walker creates objects. The database engine will retrieve the needed information from the database and produce the desired outcome in a binding table by naively evaluating the query from left to right. The second chapter entails the definitions of property graphs and related definitions such as nodes and edges. The third chapter emphasizes graph pattern matching fundamentals such as quantifiers, restrictors, and the difference between path union and multiset alternation. Chapter 4 is a continuation of 3 where path bindings are addressed, and how the final results, in theory, are obtained and returned to the user. Compared to the previous chapters, chapter 5 does a general dive into the GQL language, syntax, and clauses. Chapter 5 can be referenced on how to go from GSQL to GQL. Furthermore, some limitations are addressed and the workarounds on how to solve them. In the last chapter, the Bi7 query is evaluated with the parser. Since the view option of GQL is approved, no recent document is published after the approval. Thus, the creation of views is mostly avoided. As a solution, there is chosen to use the EXCEPT clause, and the query indeed returns the expected value, which is an  $\emptyset$ . Since GQL is new, there are many research directions that one can explore. The first direction is to research the termination of unbounded paths. Moreover, can an objective function be maximized subject to the length of the path? Another path to lean into is to verify whether fully recursive graph patterns are allowed to have multiple self-references. Lastly, since GQL is new in the future Ternary Logic Partition can be carried out for it.



# References

- [ABD<sup>+</sup>21] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savković, Michael Schmidt, Juan Sequeda, Slawek Staworko, and Dominik Tomaszuk. Pg-keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 24232436, New York, NY, USA, 2021. Association for Computing Machinery.
- [Ang18] Renzo Angles. The property graph database model. In *AMW*, 2018.
- [BFVY18] Angela Bonifati, G.H.L. Fletcher, Hannes Voigt, and N. Yakovets. *Querying graphs*. Morgan Claypool Publishers, 2018.
- [DFG<sup>+</sup>22] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 22462258, New York, NY, USA, 2022. Association for Computing Machinery.
- [DXWL20] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Aggregation support for modern graph analytics in tigergraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 377392, New York, NY, USA, 2020. Association for Computing Machinery.
- [EAL<sup>+</sup>15] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.
- [FGG<sup>+</sup>18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. Formal semantics of the language cypher. *CoRR*, abs/1802.09984, 2018.
- [GGL21] Alastair Green, Paolo Guagliardo, and Leonid Libkin. Property graphs and paths in gql: Mathematical definitions. Technical Reports TR-2021-01, Linked Data Benchmark Council (LDBC), Oct 2021.
- [Gre22] Alistair Green. Gql manifesto 2. <https://ldbcouncil.org/event/fifteenth-tuc-meeting/attachments/alastairgreengql2.0atechnicalmanifest>, 2022.
- [GSQ22a] GSQL. Ldbc snb business intelligence workload. <https://github.com/ldbc/ldbcsnbbi>, 2022.
- [GSQ22b] GSQL. Ldbc snb interactive intelligence workload. <https://github.com/ldbc/ldbcsnbinteractiveimpls>, 2022.
- [Har21] Harsh Vrajeshkumar Thakker. *On Supporting Interoperability between RDF and Property Graph Databases*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, May 2021.
- [LDB21] LDBC. *Informal working drafts*. WG3 Database Languages, 2020-2021.
- [Meg22] Ryan Meghoe. Graph query translation. <https://github.com/ryanGITC/GRAPH-QUERY-CONVERSION-GSQL-.git>, 2022.
- [Mor21] Olaf Morra. A semantics of gql a new query language for property graphs formalized. 2021.
- [Neo18] Neo4j. *GQL Scope and Features*. WG3 Database Languages, 2018.

- [RS20] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.
- [SPA<sup>+</sup>18] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. An early look at the LDBC Social Network Benchmark’s Business Intelligence workload. In *GRADES-NDA at SIGMOD/PODS*, pages 9:1–9:11. ACM, 2018.
- [W3C22a] W3C. Knuth: Computers and typesetting. <https://www.w3.org/TR/rdf11-concepts/>, 2022.
- [W3C22b] W3C. The upcoming gql standard. <https://https://www.w3.org/TR/rdf11-concepts/>, 2022.

## 8 Appendix

### 8.1 Populating Graph Database

The data used to populate the in-memory graph database is retrieved from [EAL<sup>+</sup>15] and the [SPA<sup>+</sup>18]. Although the graph of the LDBC Workbench indicates that all edges are undirected, the direction of the edges must be defined as both directed and undirected in the parser.

### 8.2 Business Intelligence Queries

For the business intelligence queries, the Bi8,10,13,14,15,16,19, and 20 could not be translated due to the phase in which GQL is at the moment, and the available options that come with it.

#### 8.2.1 BI-1

```

1 FROM BI
2 MATCH (m:Message),(mc:Comment)
3 WHERE message.creationDate < $datetime
4 AND message.content IS NOT UNKNOWN
5 CASE
6 WHEN m.length < 40 THEN 0 -- short
7     WHEN m.length < 80 THEN 1
8     WHEN m.length < 160 THEN 2
9     ELSE 3
10 END AS lengthCategory
11 Return m.year as messageYear,
12         mc as isComment,
13         lengthCategory,
14         count(message) AS messageCount,
15         sum(message.length)/toFloat (count(message)) AS averageMessageLength,
16         sum(length) AS sumMessageLength,
17         messageCount / toFloat(totalMessageCountInt) AS percentageOfMessages
18 GROUP BY messageYear,
19           isComment,
20           lengthCategory,
21 ORDER BY messageYear DESC,
22           isComment ASC,
23           lengthCategory ASC
24

```

Listing 14: Summary Posts in GQL Cypher

## 8.2.2 BI-2

```

1 SELECT m.year as messageYear, mc as isComment,
2       lengthCategory, count(message) AS messageCount,
3       sum(message.length)/toFloat(count(message)) AS averageMessageLength,
4       sum(length) AS sumMessageLength,
5       messageCount / toFloat(totalMessageCountInt) AS percentageOfMessages
6 FROM BI
7 MATCH (m:Message),(mc:Comment)
8 WHERE message.creationDate < $datetime
9 AND message.content IS NOT UNKNOWN
10 CASE
11 WHEN m.length < 40 THEN 0 -- short
12     WHEN m.length < 80 THEN 1
13     WHEN m.length < 160 THEN 2
14     ELSE 3
15 END AS lengthCategory
16 GROUP BY messageYear,
17         isComment,
18         lengthCategory,
19 ORDER BY messageYear DESC,
20         isComment ASC,
21         lengthCategory ASC
22

```

Listing 15: Summary Posts in GQL SQL

```

1 FROM BI
2 MATCH (tag:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
3 OPTIONAL MATCH (message1:Message)-[:HAS_TAG]->(tag)
4 WHERE message1.creationDate BETWEEN $Date AND $Date + 100
5 OPTIONAL MATCH (message2:Message)-[:HAS_TAG]->(tag)
6 WHERE message2.creationDate BETWEEN $date+ 100})
7 AND
8 message2.creationDateWHERE $d < $date + 200
9 Return tag.name as tag_name
10     ,count(message1) AS countWindow1
11     ,count(message2) AS countWindow2
12     ,abs(countWindow1 - countWindow2) AS difference
13 ORDER BY difference DESC,
14         tag.name ASC
15 LIMIT 100

```

Listing 16: Evolution of Tag in GQL Cypher

### 8.2.3 BI-3

```

1 SELECT    tag.name AS tag_name
2           ,count(message1) AS countWindow1
3           ,count(message2) AS countWindow2
4           ,abs(countWindow1 - countWindow2) AS difference
5 FROM BI
6 MATCH (tag:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
7 OPTIONAL MATCH (message1:Message)-[:HAS_TAG]->(tag)
8 WHERE message1.creationDate BETWEEN $Date AND $Date + 100
9 OPTIONAL MATCH (message2:Message)-[:HAS_TAG]->(tag)
10 WHERE message2.creationDate BETWEEN $date+ 100})
11 AND
12 message2.creationDate WHERE $d < $date + 200
13 ORDER BY difference DESC,
14          tag.name ASC
15 LIMIT 100

```

Listing 17: Evolution of Tag in GQL SQL

```

1 FROM BI
2 MATCH (:Country {country: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN] ,
3       (person:Person)<-[:HAS_MODERATOR]-(:f:FORUM)-[:CONTAINER_OF]
4       -> (post:Post)<-[:REPLY_OF]{0,}->(message:Message)-[:HAS_TAG]
5       ->(:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
6 RETURN f.id AS f_id,
7        f.title AS f_title,
8        f.creationDate AS f_creationDate,
9        f.ModeratorPersonId AS person_id,
10       count(DISTINCT MessageThread.MessageId) AS messageCount
11 GROUP BY f_id
12         ,f_title
13         ,f_creationDate
14         ,f_ModeratorPersonId
15 ORDER BY messageCount DESC,
16         f_id
17 LIMIT 20

```

Listing 18: Popular Topics per country in GQL Cypher

#### 8.2.4 BI-4

```

1 SELECT f.id AS f_id,
2       f.title AS f_title, f.creationDate AS f_creationDate, f.ModeratorPersonId AS person_id
3 FROM BI
4 MATCH (:Country {country: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN],
5       (person:Person)<-[:HAS_MODERATOR]-(f:FORUM)-[:CONTAINER_OF]
6       -> (post:Post)<-[:REPLY_OF]{0,}->(message:Message)-[:HAS_TAG]
7       ->(:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
8
9
10      count(DISTINCT MessageThread.MessageId) AS messageCount
11 GROUP BY f_id
12          ,f_title
13          ,f_creationDate
14          ,f_ModeratorPersonId
15 ORDER BY messageCount DESC,
16          f_id
17 LIMIT 20

```

Listing 19: Popular Topics per country in GQL SQL

```

1 FROM BI
2 MATCH (country:Country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-
3       (person:Person)<-[:HAS_MEMBER]-(forum:Forum),
4       (person:Person)<-[:HAS_MEMBER]-(topForum2:Forum)
5 WHERE forum.creationDate > $date
6 OPTIONAL MATCH SHORTEST (forum)-[:CONTAINER_OF]
7       ->(post:Post)<-[:REPLY_OF]*->(message:Message)
8       -[:HAS_CREATOR]->(person)
9 WHERE topForum2 IN topForums
10 RETURN
11      person.id AS personId,
12      person.firstName AS personFirstName,
13      person.lastName AS personLastName,
14      person.creationDate AS personCreationDate,
15      count(DISTINCT message) AS messageCount
16 ORDER BY
17      messageCount DESC,
18      person.id ASC
19 LIMIT 100

```

Listing 20: Message Creators who are top per Country in GQL Cypher

## 8.2.5 BI-5

```

1 SELECT person.id AS personId, person.firstName AS personFirstName,
2       person.lastName AS personLastName, person.creationDate AS personCreationDate,
3       count(DISTINCT message) AS messageCount
4 FROM BI
5 MATCH (country:Country)<-[:IS_PART_OF]-(city:City)<-[:IS_LOCATED_IN]
6       (person:Person)<-[:HAS_MEMBER]-(forum:Forum),
7       (person:Person)<-[:HAS_MEMBER]-(topForum2:Forum)
8 WHERE forum.creationDate > $date
9 OPTIONAL MATCH SHORTEST (forum)-[:CONTAINER_OF]
10      (post:Post)-[:REPLY_OF]-(message:Message)-
11      [:HAS_CREATOR]->(person)
12 WHERE topForum2 IN topForums // IN predicate under consensus
13 ORDER BY
14     messageCount DESC,
15     person.id ASC
16 LIMIT 100

```

Listing 21: Message Creators who are top per Country in GQL SQL

```

1 FROM BI
2 MATCH (tag:Tag {name: "Rumi"})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(p:Person)
3 OPTIONAL MATCH (message)<-[:LIKES]-(person:Person)
4 OPTIONAL MATCH (message)<-[:REPLY_OF]-(reply:Comment)
5 RETURN p.id as person_id,
6        SUM(reply) AS totalReplies,
7        SUM(lk) AS totalLikes,
8        count(message) AS messages,
9        1*messages + 2*totalReplies + 10*totalLikes AS score
10 ORDER BY
11     score DESC,
12     person_id ASC
13 LIMIT 100

```

Listing 22: Per topic Authorative User in GQL Cypher

## 8.2.6 BI-6

```

1 RETURN p.id AS person_id,
2     SUM(reply) AS totalReplies,
3     SUM(lk) AS totalLikes,
4     count(message) AS messages,
5     1*messages + 2*totalReplies + 10*totalLikes AS score
6 FROM BI
7 MATCH (tag:Tag {name: "Rumi"})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(p:Person)
8 OPTIONAL MATCH (message)<-[:LIKES]-(p:Person)
9 OPTIONAL MATCH (message)<-[:REPLY_OF]-(reply:Comment)
10 ORDER BY
11     score DESC,
12     person_id ASC
13 LIMIT 100

```

Listing 23: Per topic Authorative User in GQL Cypher

```

1 FROM BI
2 MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message)
3 -[:HAS_CREATOR]->(person:Person)
4 OPTIONAL MATCH [:REPLY_OF]-(reply:Comment)->(message)
5     <-[:LIKES]-(p:Person),
6
7     (message1)<-[:LIKES]-(person2:Person)
8 RETURN DISTINCT person.creatorPersonId
9     ,count(DISTINCT like) AS likerPerson
10 ORDER BY authorityScore DESC,
11     pl.posterPersonid ASC
12 LIMIT 100

```

Listing 24: Per topic Authorative User in GQL Cypher

```

1 SELECT DISTINCT person.creatorPersonId ,count(DISTINCT like) AS likerPerson
2 FROM BI
3 MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(person:Person)
4 OPTIONAL MATCH [:REPLY_OF]-(reply:Comment)->(message)<-[:LIKES](:Person),
5     (message1)<-[:LIKES]-(person2:Person)
6 ORDER BY authorityScore DESC,
7     pl.posterPersonid ASC
8 LIMIT 100

```

Listing 25: Per topic Authorative User in GQL SQL

### 8.2.7 BI-7



```

1 FROM BI
2 MATCH (tag:Tag {name:$tagname} )<-[:HAS_TAG]-(m:Message),
3       (m)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)
4 RETURN rTag.name AS rTag_name,
5        count(*) AS count
6 GROUP BY rTag_name
7 ORDER BY count DESC
8         ,rTag_name
9 LIMIT 100
10 EXCEPT ALL
11 FROM BI
12 MATCH (c1:Comment)~[:HAS_TAG]~(tag1:Tag)
13 Return tag1 as rTag_name

```

Listing 26: Topics related to one another in GQL Cypher

```

1 SELECT rTag.name AS rTag_name,
2        count(*) AS count
3 FROM BI
4 MATCH (tag:Tag {name:$tagname} )<-[:HAS_TAG]-(
5       (m:Message)<-[:REPLY_OF]-(c:Comment)~
6       [:HAS_TAG]~>(rTag:Tag)
7 GROUP BY rTag_name
8 ORDER BY count DESC
9         ,rTag_name
10 LIMIT 100

```

Listing 27: Topics related to one another in GQL SQL

## 8.2.8 BI-9

```

1 FROM BI
2 MANDATORY MATCH (person:Person)<-[:HAS_CREATOR]-(post:Post)<-[:REPLY_OF]-(reply:Message)
3 WHERE post.creationDate BETWEEN $startDate AND $endDate
4 AND reply.creationDate BETWEEN $startDate AND $endDate
5 RETURN person_id ,person_firstName,person_lastName
6        ,count(DISTINCT post) AS threadCount
7        ,count(DISTINCT reply) AS messageCount
8 ORDER BY
9        messageCount DESC,
10       person.id ASC
11 LIMIT 100

```

Listing 28: Central Person for a Tag in GQL Cypher

```

1 SELELCT person_id ,person_firstName,person_lastName
2     ,count(DISTINCT post) AS threadCount
3     ,count(DISTINCT reply) AS messageCount
4 FROM BI
5 MANDATORY MATCH (person:Person)<-[:HAS_CREATOR]-(post:Post)<-[:REPLY_OF]-(reply:Message)
6 WHERE post.creationDate BETWEEN $startDate AND $endDate
7 AND reply.creationDate BETWEEN $startDate AND $endDate
8 ORDER BY
9     messageCount DESC,
10    person.id ASC
11 LIMIT 100

```

Listing 29: Central Person for a Tag in GQL SQL

### 8.2.9 BI-11

```

1 FROM BI
2 MATCH ANY ACYCLIC (country:Country {name: $country}),
3     (person1:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
4     (person2:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
5     (person3:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
6     (person1)-[k1:KNOWS]-(person2)-[k2:KNOWS]-(person3)-[k3:KNOWS]-(person1)
7 WHERE person1.id BETWEEN person2.id AND person1
8     AND k1 creationDate BETWEEN $startDate AND $endDate
9     AND k2 creationDate BETWEEN $startDate AND $endDate
10    AND k2 creationDate BETWEEN $startDate AND $endDate
11 RETURN count(*) AS count

```

Listing 30: Triangles of Friend in GQL Cypher

```

1 SELECT count(*) AS count
2 FROM BI
3 MATCH ANY ACYCLIC (country:Country {name: $country}),
4     (person1:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
5     (person2:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
6     (person3:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
7     (person1)-[k1:KNOWS]-(person2)-[k2:KNOWS]-(person3)-[k3:KNOWS]-(person1)
8 WHERE person1.id BETWEEN person2.id AND person1
9     AND k1 creationDate BETWEEN $startDate AND $endDate
10    AND k2 creationDate BETWEEN $startDate AND $endDate
11    AND k3 creationDate BETWEEN $startDate AND $endDate
12 RETURN count(*) AS count

```

Listing 31: Triangles of Friend in GQL SQL

### 8.2.10 BI-12

```

1 FROM BI
2 MATCH (person:Person)<-[:HAS_CREATOR]-(message:Message)-[:REPLY_OF]*->(post:Post)
3 WHERE post.language = "ar" OR post.language = "hu"
4 AND message.length < 20
5 AND message.creationDate > $startDate
6 RETURN count(message) AS messageCount,
7        count(person) AS personCount
8 ORDER BY
9     personCount DESC,
10    messageCount DESC

```

Listing 32: Number of Posts per person in GQL Cypher

```

1 SELECT count(message) AS messageCount, count(person) AS personCount
2 FROM BI
3 MATCH (person:Person)<-[:HAS_CREATOR]-(message:Message)
4      [][:REPLY_OF]*->(post:Post)
5      WHERE post.language = "ar" OR post.language = "hu"
6 AND message.length < 20
7 AND message.creationDate > $startDate
8 ORDER BY
9     personCount DESC,
10    messageCount DESC

```

Listing 33: Number of Posts per person in GQL SQL

## 8.2.11 BI-17

```

1 FROM BI
2 MATCH SHORTEST
3     (tag:Tag {name: $tag}),
4     (person1:Person)<-[:HAS_CREATOR]-(message1:Message)-[:REPLY_OF]
5     *->(post1:Post)<-[:CONTAINER_OF]-(forum1:Forum),
6     (message1)-[:HAS_TAG]->(tag),
7     (forum1)<-[:HAS_MEMBER]->(person2:Person)<-[:HAS_CREATOR]-(comment:Comment)
8     -[:HAS_TAG]->(tag),
9     (forum1)<-[:HAS_MEMBER]->(person3:Person)<-[:HAS_CREATOR]-(message2:Message), (comment)-[:HAS_TAG]->(tag),
10    (message2)-[:HAS_TAG]->(tag),
11    (forum2)-[:HAS_MEMBER]->(person1)
12 WHERE forum1 <> forum2
13 AND message2.creationDate > message1.creationDate + duration({hours: $delta})
14 RETURN person1.id, count(DISTINCT message2) AS messageCount
15 ORDER BY messageCount DESC, person1.id ASC
16 LIMIT 10

```

Listing 34: Analysis of Information flow in GQL Cypher

```

1 SELECT person1.id, count(DISTINCT message2) AS messageCount
2 FROM BI
3 MATCH SHORTEST
4   (tag:Tag {name: $tag}),
5   (person1:Person)<-[:HAS_CREATOR]-(message1:Message)-[:REPLY_OF]
6   *->(post1:Post)<-[:CONTAINER_OF]
7   -(forum1:Forum), (message1)-[:HAS_TAG]->(tag), (forum1) <-[:HAS_MEMBER]->(person2:Person)
8   <-[:HAS_CREATOR]-(comment:Comment)
9   -[:HAS_TAG]->(tag),
10  (forum1)<-[:HAS_MEMBER]
11  ->(person3:Person)<-[:HAS_CREATOR]-(message2:Message),
12
13  (comment)-[:HAS_TAG]->(tag),
14  (message2)-[:HAS_TAG]->(tag),
15  (forum2)-[:HAS_MEMBER]->(person1)
16 WHERE forum1 <> forum2
17       AND message2.creationDate > message1.creationDate + duration({hours: $delta})
18  (forum2)-[:HAS_MEMBER]->(person1)
19 ORDER BY messageCount DESC, person1.id ASC
20 LIMIT 10

```

Listing 35: Analysis of Information flow in GQL SQL

### 8.2.12 BI-18

```

1 FROM LDBC_SNB
2 MATCH   (tag:Tag {name: $tag})<-[:HAS_INTEREST]-(person1:Person)
3         -[:KNOWS]-(commonFriend:Person)-[:HAS_INTEREST]->(tag)
4 OPTIONAL MATCH (commonFriend:Person)-[:KNOWS]-(person2:Person),
5               (person1)-[:KNOWS]-(person2)
6 WHERE person1 <> person2
7 RETURN person1.id AS idPerson1
8         ,person2.id AS idPerson2
9         ,count(DISTINCT mutualFriend) AS mutualFriendCount
10 ORDER BY mutualFriendCount DESC
11         ,person1Id ASC
12         ,person2Id ASC
13 LIMIT 20

```

Listing 36: Recommendation of Friends in GQL Cypher

```

1 SELECT person1.id AS idPerson1
2       ,person2.id AS idPerson2
3       ,count(DISTINCT mutualFriend) AS mutualFriendCount
4 FROM BI
5 MATCH (tag:Tag {name: $tag})<-[:HAS_INTEREST]-(person1:Person)
6       -[:KNOWS]-(commonFriend:Person)-[:HAS_INTEREST]->(tag)
7 OPTIONAL MATCH (commonFriend:Person)-[:KNOWS]
8               -(person2:Person), (person1)-[:!KNOWS]-(person2)
9 WHERE person1 <> person2
10 ORDER BY mutualFriendCount DESC
11         ,person1Id ASC
12         ,person2Id ASC
13 LIMIT 20

```

Listing 37: Recommendation of Friends in GQL SQL

## 8.3 Interactive Queries

In here also due to the limited function and options that are available in GQL the following queries were only able to be converted

### 8.3.1 Interactive 2

```

1 FROM INA
2 MATCH (P:Person {id: $personId })-[:KNOWS]-(f:Person)<-[:HAS_CREATOR]-(m:Message)
3 WHERE message.creationDate <= $maxDate
4 RETURN
5     f.id AS personId,
6     f.firstName AS personFirstName,
7     f.lastName AS personLastName,
8     m.id AS postID,
9     coalesce(m.content,m.imageFile) as postOrCommentContent
10    m.creationDate AS postDate
11 ORDER BY
12     postOrCommentContent DESC,
13     toInteger(postOrCommentId) ASC
14 LIMIT 20

```

Listing 38: Friend's Recent Messages in GQL Cypher

```

1 SELECT f.id AS personId,f.firstName AS personFirstName,f.lastName AS personLastName
2      ,m.id AS postID,coalesce(m.content,m.imageFile) as postOrCommentContent,
3      m.creationDate AS postDate
4 FROM INA
5 MATCH (P:Person {id: $personId })-[:KNOWS]-(f:Person)<-[:HAS_CREATOR]-(m:Message)
6 WHERE message.creationDate <= $maxDate
7 ORDER BY
8      postOrCommentContent DESC,
9      toInteger(postOrCommentId) ASC
10 LIMIT 20

```

Listing 39: Friend's Recent Messages in GQL SQL

### 8.3.2 Interactive Query 4

```

1 FROM INA
2 MATCH (p:Person {id: $personId })-[:KNOWS]-(f:Person) <-[:HAS_CREATOR]-(post:Post)-[:HAS_TAG]-(tag)
3 CASE $tag :
4     WHEN $endDate > post.creationDate >= $startDate THEN 1
5     ELSE 0
6     END AS valid,
7     CASE
8         WHEN $startDate > post.creationDate THEN 1
9         ELSE 0
10        END AS invalid
11 WHERE countOfPost>0 AND invalidCountOfPost=0
12 RETURN tag.name AS tagName,
13        ,sum(valid) AS countOfPost
14        ,sum(invalid) AS invalidCountOfPost
15 ORDER BY postCount DESC
16        ,tagName ASC
17 LIMIT 10

```

Listing 40: Topics in GQL Cypher

```

1 SELECT tag.name AS tagName,sum(valid) AS countOfPos,sum(inValid) AS inValidCountOfPost
2 FROM INA
3 MATCH (p:Person {id: $personId })-[:KNOWS]-(f:Person) <-[:HAS_CREATOR]-(post:Post)-[:HAS_TAG]->(tag)
4 CASE $tag :
5     WHEN $endDate > post.creationDate >= $startDate THEN 1
6     ELSE 0
7     END AS valid,
8     CASE
9         WHEN $startDate > post.creationDate THEN 1
10        ELSE 0
11        END AS inValid
12 WHERE countOfPost>0 AND inValidCountOfPost=0
13 ORDER BY postCount DESC
14         ,tagName ASC
15 LIMIT 10

```

Listing 41: Topics in GQL SQL

### 8.3.3 Interactive Query 6

```

1 FROM INA
2 MATCH (p:Person { id: $personId })-[:KNOWS]-(friend:Person){1,1}
3 WHERE person <> f
4 MATCH (friend)<-[:membership:HAS_MEMBER]-(forum),
5 MATCH (friend)<-[:HAS_CREATOR]-(post)<-[:CONTAINER_OF]-(forum)
6 WHERE membership.creationDate > $minDate
7     AND friend IN otherPersons // IN needs to be specified
8
9 RETURN
10     forum.title AS forumName,
11     count(post) AS postCountpostCount
12 ORDER BY
13     postCount DESC,
14     forum.id ASC
15 LIMIT 20

```

Listing 42: Groups in GQL Cypher



```

1 SELECT forum.title AS forumName,count(post) AS postCountpostCount
2 FROM INA
3 MATCH (p:Person { id: $personId })-[:KNOWS]-(friend:Person){1,1}
4 WHERE person <> f
5 MATCH (friend)<-[membership:HAS_MEMBER]-(forum),(friend)<-[:HAS_CREATOR]-(Post)
6 <-[:CONTAINER_OF]-(Forum)
7 WHERE membership.creationDate > $minDate
8     AND friend IN otherPersons // IN needs to be specified
9 ORDER BY
10     postCount DESC,
11     forum.id ASC
12 LIMIT 20

```

Listing 43: Groups in GQL SQL

### 8.3.4 Interactive 7

```

1 FROM INA
2 MATCH (m:Message {id: $messageId })<-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p:Person)
3 OPTIONAL MATCH (m)-[:HAS_CREATOR]->(a:Person)-[r:KNOWS]-(p)
4 RETURN c.id AS commentId,
5         c.content AS Content,
6         c.creationDate AS CreationDate,
7         p.id AS AuthorId,
8         p.firstName AS replyFirstName,
9         p.lastName AS replyLastName,
10        CASE r
11            WHEN null THEN false
12            ELSE true
13        END AS KnowsOriginalMessageAuthor
14 ORDER BY creationDate DESC
15         ,AuthorId

```

Listing 44: Likers in GQL Cypher

```

1 SELECT c.id AS commentId,c.content AS Content, c.creationDate AS CreationDate,
2       p.id AS AuthorId,p.firstName AS replyFirstName,p.lastName AS replyLastName,
3       CASE r
4         WHEN null THEN false
5         ELSE true
6       END AS KnowsOriginalMessageAuthor
7 FROM INA
8 MATCH (m:Message {id: $messageId })<-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p:Person)
9 OPTIONAL MATCH (m)-[:HAS_CREATOR]->(a:Person)-[r:KNOWS]-(p)
10 ORDER BY creationDate DESC
11         ,AuthorId

```

Listing 45: Likers in GQL SQL

### 8.3.5 Interactive 8

```

1 FROM INA
2 MATCH (p1:Person {id: $personId})<-[:HAS_CREATOR]-(c:Message)
3 <-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p2:Person)
4 RETURN
5     p1.id AS personId,
6     p1.firstName AS personFirstName,
7     p1.lastName AS personLastName,
8     c.creationDate AS commentCreationDate,
9     c.id AS commentId,
10    c.content AS content
11 ORDER BY
12     commentCreationDate DESC,
13     commentId ASC
14 LIMIT 20

```

Listing 46: Reply in GQL Cypher

```

1 SELECT p1.id AS personId, p1.firstName AS personFirstName,
2       p1.lastName AS personLastName, c.creationDate AS commentCreationDate,
3       c.id AS commentId, c.content AS content
4 FROM INA
5 MATCH (p1:Person {id: $personId})<-[:HAS_CREATOR]-(:Message)
6      <-[:REPLY_OF]-(:Comment)-[:HAS_CREATOR]->(p2:Person)
7 ORDER BY
8       commentCreationDate DESC,
9       commentId ASC
10 LIMIT 20

```

Listing 47: Reply in GQL SQL

### 8.3.6 Interactive 9

```

1 FROM INA
2 MATCH (p1:Person {id: $personId })-[:KNOWS]-(:p2:Person){1,2}
3 WHERE f<>p1 // in GQL parser not necessary, we add distinct values
4 MATCH (f) <-[:HAS_CREATOR]-(:m:Message)
5       WHERE m.creationDate < $maxDate
6 RETURN
7       f.id AS personId,
8       f.firstName AS FirstName,
9       f.lastName AS LastName,
10      m.id AS commentId,
11      coalesce(m.content,m.imageFile) AS Content,
12      m.creationDate AS contentCreationDate
13 ORDER BY
14      contentCreationDate DESC,
15      message.id ASC
16 LIMIT 20

```

Listing 48: Messages of Friends by Friends in GQL Cypher

```

1 SELECT f.id AS personId,f.firstName AS FirstName,
2       f.lastName AS LastName,m.id AS commentId,
3       coalesce(m.content,m.imageFile) AS Content,
4       m.creationDate AS contentCreationDate
5 FROM INA
6 MATCH (p1:Person {id: $personId })-[:KNOWS]-(p2:Person){1,2}
7 WHERE f<>p1 // in GQL parser not necessary, we add distinct values
8 MATCH (f) <-[:HAS_CREATOR]-(m:Message)
9       WHERE m.creationDate < $maxDate
10 ORDER BY
11       contentCreationDate DESC,
12       message.id ASC
13 LIMIT 20

```

Listing 49: Messages of Friends by Friends in GQL SQL

### 8.3.7 Interactive 11

```
1 FROM INA
2 MATCH (p1:Person {id: $personId })-[:KNOWS]-(p2:Person){1,2}
3 WHERE person <> p2 //not necessary
4 MATCH (:Country{name: $countryName })-[:IS_LOCATED_IN]-(
5     friend)-[workAt:WORK_AT]-(company:Company)
6 WHERE workAt.workFrom < $workFromYear
7 RETURN
8     f.id AS personId,
9     f.firstName AS personFirstName,
10    f.lastName AS personLastName,
11    company.name AS organizationName,
12    workAt.workFrom AS organizationWorkFromYear
13 ORDER BY
14     organizationWorkFromYear ASC,
15     toInteger(personId) ASC,
16     organizationName DESC
17 LIMIT 10
```

Listing 50: Referral of Jobs in GQL Cypher

```
1 SELECT f.id AS personId,f.firstName AS personFirstName,
2     f.lastName AS personLastName,company.name AS organizationName,
3     workAt.workFrom AS organizationWorkFromYear
4 FROM INA
5 MATCH (p1:Person {id: $personId })-[:KNOWS]-(p2:Person){1,2}
6 WHERE person <> p2 //
7 MATCH (:Country{name: $countryName })-[:IS_LOCATED_IN]-(friend)-[workAt:WORK_AT]-(company:Company)
8 WHERE workAt.workFrom < $workFromYear
9 ORDER BY
10     organizationWorkFromYear ASC,
11     toInteger(personId) ASC,
12     organizationName DESC
13 LIMIT 10
```

Listing 51: Referral of Jobs in GQL SQL

### 8.3.8 Interactive 12

```
1 FROM INA
2 MATCH SHORTEST (baseTagClass:TagClass)<-[:HAS_TYPE|IS_SUBCLASS_OF]*-(tag1:Tag) //change accordingly
3 WHERE tag.name = $tagClassName OR baseTagClass.name = $tagClassName
4 MATCH (:Person {id: $personId })-[:KNOWS]-(friend:Person)
5 <-[:HAS_CREATOR]-(comment:Comment)-[:REPLY_OF]->(:Post)-[:HAS_TAG]->(tag:Tag)
6 WHERE tag.id == tag1.id //alternative for IN
7 RETURN
8     friend.id AS personId,
9     friend.firstName AS personFirstName,
10    friend.lastName AS personLastName,
11    COLLECT(DISTINCT tag.name) AS tagNames,
12    COUNT(DISTINCT comment) AS replyCount
13 ORDER BY
14     replyCount DESC,
15     toInteger(personId) ASC
16 LIMIT 20
```

Listing 52: Experts in GQL Cypher

```
1 SELECT friend.id AS personId,
2         friend.firstName AS personFirstName,
3         friend.lastName AS personLastName,
4         COLLECT(DISTINCT tag.name) AS tagNames,
5         COUNT(DISTINCT comment) AS replyCount
6 FROM INA
7 MATCH SHORTEST (baseTagClass:TagClass)<-[:HAS_TYPE|IS_SUBCLASS_OF]*-(tag1:Tag) //change accordingly
8 WHERE tag.name = $tagClassName OR baseTagClass.name = $tagClassName
9 MATCH (:Person {id: $personId })-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-(comment:Comment)
10 -[:REPLY_OF]->(:Post)-[:HAS_TAG]->(tag:Tag)
11 WHERE tag.id == tag1.id //alternative for IN ?
12 ORDER BY
13     replyCount DESC,
14     toInteger(personId) ASC
15 LIMIT 20
```

Listing 53: Experts in GQL SQL

### 8.3.9 Interactive 13

```
1 FROM INA
2 MATCH SHORTEST p = shortest((p1)-[:KNOWS]-(p2)){*}
3 RETURN
4     p AS shortestPathLength
```

Listing 54: Shortest Path in GQL Cypher

```
1 SELECT p AS shortestPathLength
2 FROM INA
3 MATCH SHORTEST p = shortest((p1)-[:KNOWS]-(p2)){*}
```

Listing 55: Shortest Path in GQL SQL

### 8.3.10 Interactive short Query 1

```
1 FROM INA
2 MATCH (p1:Person {id: $personId })-[:IS_LOCATED_IN]->(c:City)
3 RETURN
4     p1.firstName AS firstName,
5     p1.lastName AS lastName,
6     p1.birthday AS birthday,
7     p1.locationIP AS locationIP,
8     p1.browserUsed AS browserUsed,
9     p1.id AS cityId,
10    p1.gender AS gender,
11    p1.creationDate AS creationDate
```

Listing 56: Person's Profile in GQL Cypher

```
1 SELECT p1.firstName AS firstName,
2        p1.lastName AS lastName,
3        p1.birthday AS birthday,p1.locationIP AS locationIP,
4        p1.browserUsed AS browserUsed
5        ,p1.id AS cityId,p1.gender AS gender,p1.creationDate AS creationDate
6 FROM INA
7 MATCH (p1:Person {id: $personId })-[:IS_LOCATED_IN]->(c:City)
```

Listing 57: Person's Profile in GQL SQL



### 8.3.11 Interactive short Query 2

```
1 FROM INA
2 MATCH (:Person {id: $personId})<-[:HAS_CREATOR]-(:message)
3 OPTIONAL MATCH (message)-[:REPLY_OF]->{0,0}(post:Post)-[:HAS_CREATOR]->(person)
4 Return
5     message,
6     message.id AS messageId,
7     message.creationDate AS messageCreationDate
8     ,coalesce(message.imageFile,message.content) AS messageContent
9     ,post.id AS postId,
10    ,person.id AS personId,
11    ,person.firstName AS FirstName,
12    ,person.lastName AS LastName
13 ORDER BY messageCreationDate DESC
14         ,messageId ASC
```

Listing 58: Recent Message in GQL Cypher

```
1 SELECT message, message.id AS messageId,message.creationDate AS messageCreationDate
2     ,coalesce(message.imageFile,message.content) AS messageContent
3     ,post.id AS postId,person.id AS personId,person.firstName AS FirstName,
4     ,person.lastName AS LastName
5 FROM INA
6 MATCH (:Person {id: $personId})<-[:HAS_CREATOR]-(:message)
7 OPTIONAL MATCH (message)-[:REPLY_OF]->{0,0}(post:Post)-[:HAS_CREATOR]->(person)
8 ORDER BY messageCreationDate DESC
9         ,messageId ASC
```

Listing 59: Recent Message in GQL SQL

### 8.3.12 Interactive short Query 3

```
1 FROM INA
2 MATCH (n:Person {id: $personId })-[r:KNOWS]-(f:friend)
3 RETURN
4     f.id AS personId,
5     f.firstName AS firstName,
6     f.lastName AS lastName,
7     r.creationDate AS friendshipDate
8 ORDER BY
9     friendshipDate DESC,
10    personId ASC
```

Listing 60: in GQL Cypher

```
1 SELECT f.id AS personId,f.firstName AS firstName,
2        f.lastName AS lastName,r.creationDate AS friendshipDate
3 FROM INA
4 MATCH (n:Person {id: $personId })-[r:KNOWS]-(f:friend)
5 ORDER BY
6     friendshipDate DESC,
7     personId ASC
```

Listing 61: Friend's Person in GQL SQL

### 8.3.13 Interactive short Query 4

```
1 FROM INA
2 MATCH (message:Message {id: $messageId })
3 RETURN
4     message.creationDate as messageCreationDate,
5     COALESCENCE (m.content, m.imageFile) as messageContent
```

Listing 62: Content in GQL Cypher

```
1 SELECT message.creationDate as messageCreationDate,
2     COALESCENCE (m.content, m.imageFile) as messageContent
3 FROM INA
4 MATCH (message:Message {id: $messageId })
```

Listing 63: Content in GQL SQL

### 8.3.14 Interactive Short Query 5

```
1 FROM INA
2 MATCH (m:Message {id: $messageId })-[:HAS_CREATOR]->(p1:Person)
3 RETURN
4     p1.id AS personId,
5     p1.firstName AS firstName,
6     p1.lastName AS lastName
```

Listing 64: Message Creator in GQL Cypher

```
1 SELECT p1.id AS personId,p1.firstName AS firstName,
2         p1.lastName AS lastName
3 FROM INA
4 MATCH (m:Message {id: $messageId })-[:HAS_CREATOR]->(p1:Person)
```

Listing 65: Message Creator in GQL SQL

### 8.3.15 Interactive Short Query 6

```
1 FROM INA
2 MATCH SHORTEST (m:Message {id: $messageId })-[:REPLY_OF]*->
3   (p:Post)<-[:CONTAINER_OF]-(f:Forum)
4   -[:HAS_MODERATOR]->(p1:Person)
5 RETURN
6   f.id AS forumId,
7   f.title AS forumTitle,
8   p1.id AS moderatorId,
9   p1.firstName AS FirstName,
10  p1.lastName AS LastName
```

Listing 66: Message's Forum in GQL Cypher

```
1 SELECT f.id AS forumId
2       ,f.title AS forumTitle
3       ,p1.id AS moderatorId
4       ,p1.firstName AS FirstName
5       ,p1.lastName AS LastName
6 FROM INA
7 MATCH SHORTEST (m:Message {id: $messageId })-[:REPLY_OF]*->
8   (p:Post)<-[:CONTAINER_OF]-(f:Forum)-
9   [:HAS_MODERATOR]->(p1:Person)
```

Listing 67: Message's Forum in GQL SQL

### 8.3.16 Interactive Short Query 7

```
1 FROM INA
2 MATCH (m:Message {id: $messageId })<-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p:Person)
3 OPTIONAL MATCH (m)-[:HAS_CREATOR]->(p2:Person)-[r:KNOWS]-(p1)
4 RETURN c.id AS commentId,
5         c.content AS content,
6         c.creationDate AS creationDate,
7         p.id AS AuthorId,
8         p.firstName AS AuthorFirstName,
9         p.lastName AS AuthorLastName,
10 CASE r:
11     WHEN UNKNOWN THEN false
12     ELSE true
13     END AS replyAuthorKnowsOriginalMessageAuthor
14 ORDER BY commentCreationDate DESC, AuthorId
```

Listing 68: Reply in GQL Cypher

```
1 SELECT c.id AS commentId,
2         c.content AS content,
3         c.creationDate AS creationDate,
4         p.id AS AuthorId,
5         p.firstName AS AuthorFirstName,
6         p.lastName AS AuthorLastName,
7 CASE r:
8     WHEN UNKNOWN THEN false
9     ELSE true
10    END AS replyAuthorKnowsOriginalMessageAuthor
11 FROM LDBC_SNBI
12 MATCH (m:Message {id: $messageId })<-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p:Person)
13 OPTIONAL MATCH (m)-[:HAS_CREATOR]->(p2:Person)-[r:KNOWS]-(p1)
14 ORDER BY commentCreationDate DESC, AuthorId
```

Listing 69: Reply in GQL SQL

### 8.3.17 Interactive Update 2

```
1      From INA
2      MATCH (person:Person {id: $personId}), (comment:Comment {id: $commentId})
3      CREATE (person)-[:LIKES {creationDate: $creationDate}]->(comment)
```

### 8.3.18 Interactive Update 3

```
1      From INA
2      MATCH (person:Person {id: $personId}), (comment:Comment {id: $commentId})
3      CREATE (person)-[:LIKES {creationDate: $creationDate}]->(comment)
```

### 8.3.19 Interactive Update 4

```
1      FROM INA
2      MATCH (message:Message {id: $messageId })
3      RETURN
4          message.creationDate as messageCreationDate,
5          COALESCENCE (m.content, m.imageFile) as messageContent
```

Listing 70: Content in GQL Cypher

```
1      SELECT message.creationDate as messageCreationDate,
2              COALESCENCE (m.content, m.imageFile) as messageContent
3      FROM INA
4      MATCH (message:Message {id: $messageId })
```

Listing 71: Content in GQL SQL

### 8.3.20 Interactive update 8

```
1      MATCH (p1:Person {id: $person1Id}), (p2:Person {id: $person2Id})
2      CREATE (p1)-[:KNOWS {creationDate: $creationDate}]->(p2)
```