Linked Database Council
Social Network Bench
GQL and SQL/PGX

Ryan Meghoe
email bluer.a.meghoe@student.tue.nl

August 1, 2022

Finalized: Chapters 1 -5 (Final Chapters)
Chapter 6 and afterward in draft-version
To Do List
Fix Appendix

-
  - Add missing queries
  - Write tables out for the interactive queries and label them
  - adjust queries so they fix in the box, and clean up in the report
- Write chapters 6 and 7
  - chapter 7: Discussion and Conclusion
  - Chapter 6: Validity Testing
    * Load and Test Graph in Parser
    * Test a subset of Queries

Internship(2IMC10) Internship Final Report

Faculty of Mathematics and Computer Science

Eindhoven University of Technology (TU/e)

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

1 Introduction

Representation of sophisticated network structured data is possible through graphs.Graphs are used in social networks, biology [DFG+22], and chemistry[ABD+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 [W3C] or PropertyGraphs [BFVY18] . In the context of enterprise data management, many current graph database systems offer support property graphs, such Neptune, Neo4j , TigerGraph,Oracle Server[DFG+22]. A property graph is a multigraph where nodes and edges have labels and properties. 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. GQL is composed of other Graph Languages, such as a part of SQL named SQL/PGQ [DFG+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 on a single dataset. The Social Network Benchmark's Interactive workload is the first one and focuses on transnational 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. In [Mor]the formal semantics is defined for GQL, and a naive GLQ parser is built. In this project, the previously realised 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 [GSQa] [GSQb] 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.

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

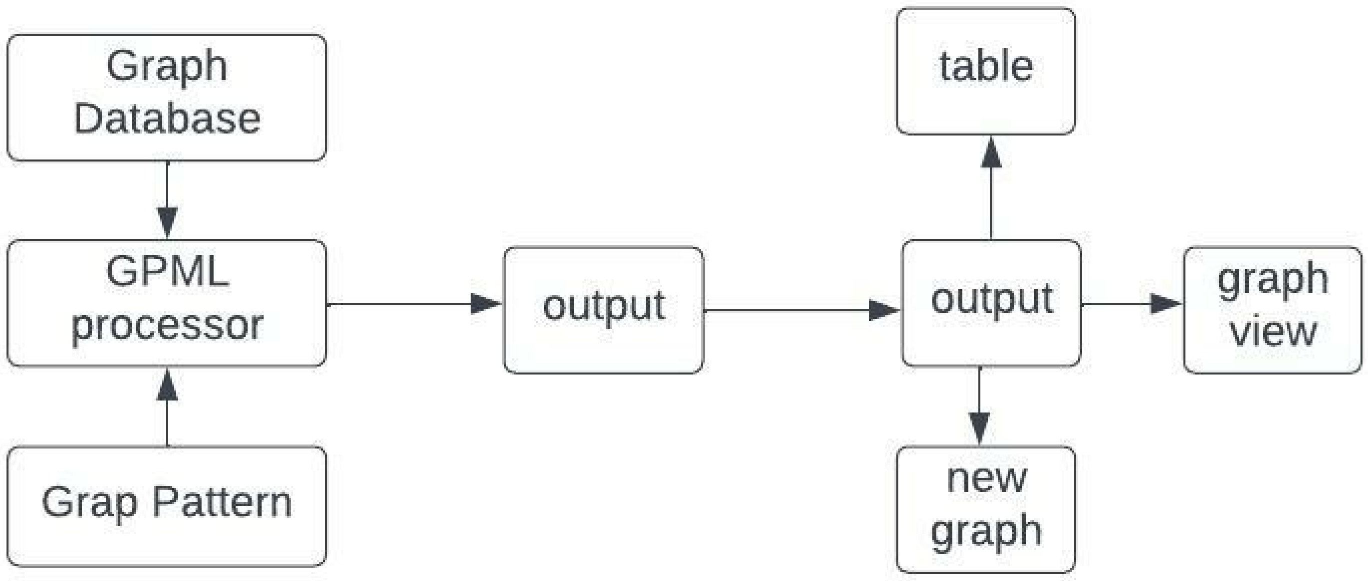

Figure 1: Conceptual Diagram of GQL and GPML

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 set() is not empty, and the values contained within that set are distinct, then the set(v_m,....,v_m) is a value
4. In contrast to a multiset, the same condition applies with distinct values excluded from the conditions.
5. A map() is considered a value if k_1....k_m are distinct.
6. A node identifier n with a 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 path(n_1, e_1, n_2, . . . , n_{m-1}, 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+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+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 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,λ,σ) and :
1. N is a finite set of vertices)
2. E is a finite set of edges
3. $\rho : E \rightarrow (N \times N\ )$ is a total function that combines each edge in E with a pair of vertices in N
4. $: (N \cup E) \rightarrow SET{+}(L)$ is a partial function that combines a vertices or edge with a set of labels from L.
5. $\sigma : (N \cup E){\times}\ P \rightarrow SET{+}(V)$ is a partial function that combines nodes or edges with properties, and for each property it assigns a set of values from V.
Bonafeti Et al further state that nodes and edges are pairwise independent.

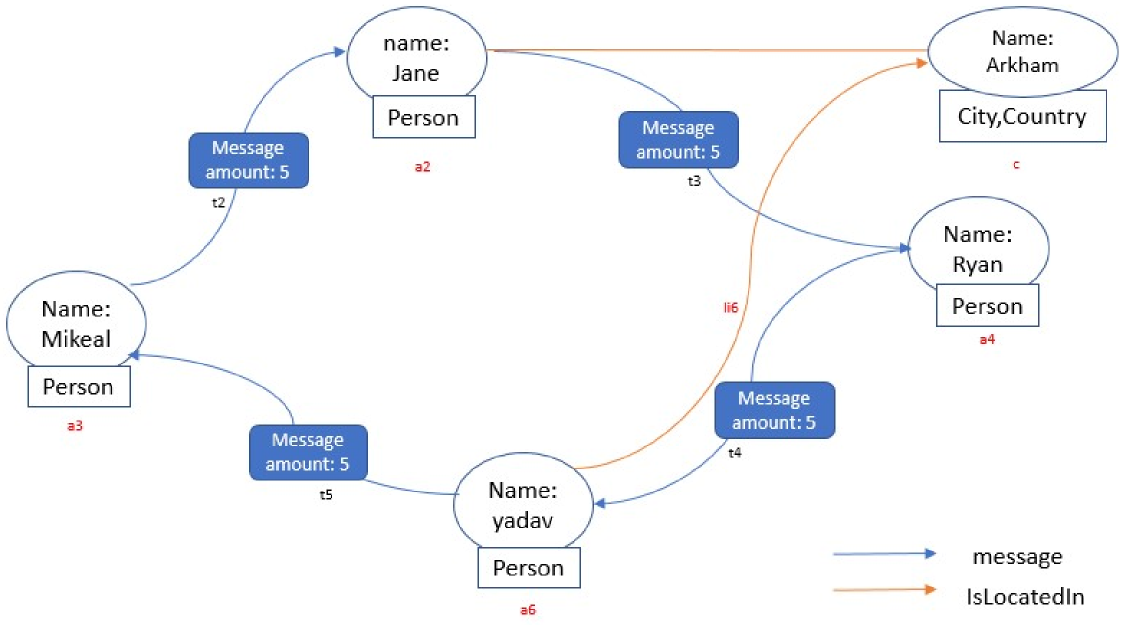

Figure 2: Propertygraph

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 [Mor] 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= $(a_1 : v_1, \ . \ . \ . \ , a_n : v_n)$. Out of the name fields we have $a_1, \ . \ . \ . \ , a_n \in A_2$, which are distinct names, and $v_1, \ . \ . \ . \ , v_n \in V$ 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 ·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,..., 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 δ: 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 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 T3becomes (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+22]uses the acronym GPML to describe the pattern matching language. The GPML language that 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 χ and is a triple of (a,L,P) and :*

- *$a \in A \cup$ {nil} is an optional name.*
- *$L \subset \varnothing \cup \{L_1,...,L_n\}$, 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+18]**Relationship Pattern** An relationship pattern ρ 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.*
- *$L \subset \varnothing \cup \{L_1,...,L_n\}$ 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 rigth

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 2there 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->country1), and the second is (c->country2).As third path, which is not a duplicate, the path (c-> 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. Node 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 equi-joins are 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 equivalents 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 grap G and the path pattern π

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+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.

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

Another use of restrictors and selectors is to ensure the termination of a GPML. Morever 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 labeleled 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+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.

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

Table 3: Selectors and their description

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 } $\in \mathbb{N}$

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.

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

Listing 1: Code example retrieved from the property graph in []

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,}. Thus, the pattern is rewritten to:

MATCH TRAIL (p **WHERE** p.person = 'Ryan')
[()-[m: Message **WHERE** count(m.message)->20]-> ()] {1,0}
(p) [()-[:isLocatedIn]->(place:City) |
() -[:isLocatedIn]->(c:Country)]

Do note that there are three empty nodes(anonymous) added compared to the initial query in 112. Now the pattern is rewritten by adding a new variable with an index, $\square_x$. The pattern rewritten becomes :

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

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 (|).

(p **WHERE** p.person = 'Ryan')
($\square_i^1$)-[m^1 : Message **WHERE** count(m.message)-> 20]-> ($\square_{ii}^1$) (p)
($\square_{iii}$)-[$-_i$:isLocatedIn]->(place:City)

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 $\theta_{n,country}$

(p **WHERE** p.person = 'Ryan')
($\square_i^1$)-[m^1 : Message **WHERE** count(m^1.message)-> 20]-> ($\square_{ii}^1$)
($\square_i^2$)-[m^2 : Message **WHERE** count(m^2.message)-> 20]-> ($\square_{ii}^2$)
($\square_i^n$)-[m^n : Message **WHERE** count(m^n.message)-> 20]-> ($\square_{ii}^n$)
(p)
($\square_{iii}$)-[$-_i$:isLocatedIn]->(place:City)

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.

(p **WHERE** p.person = 'Ryan')
-[m^1 : Message **WHERE** count(m^1.message)-> 20]-> ($\square_{ii}^1$)
-[m^2 : Message **WHERE** count(m^2.message)-> 20]-> ($\square_{ii}^2$)
-[m^{n-1} : Message **WHERE** count(m^{n-1}.message)-> 20]-> ($\square_{ii}^{n-1}$)
-[m^n : Message **WHERE** count(m^n.message)-> 20]-> ($\square_{ii}^n$)
(p)
-[$-_i$:isLocatedIn]->(place:City)

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:

a	b^1	$\square_{ii}^1$
a4	t4	a6

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.

(p **WHERE** p.name = 'Ryan')
-[b^1: Transfer WHERE b^1.count(message)>20]-> ($\square_{ii}^1$)

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:

$\square_{ii}^1$	b^2	$\square_{ii}^2$
a6	t5	a3
a3	t2	a2
a2	t3	a4

(4 more)

$\square_{ii}^1$	b^2	$\square_{ii}^2$
a6	t5	a3
a3	t2	a2
a2	t3	a4

(4 more)

$\square_{ii}^3$	b^4	**a**
a6	t5	a3
a3	t2	a2
a2	t3	a2

(4 more)

a	$-_i$	c
a4	li4	c2
a6	li4	c2
a3	li3	c1

(3 more)

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 :

a	b^1	$\square^1_{ii}$	b^2	$\square^2_{ii}$	b^3	$\square^3_{ii}$	b^4	a	$-_i$	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 [Mor], 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 The query semantics

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:

```
FROM graph
MATCH pattern1,pattern2,pattern3
RETURN variables,results
```

Listing 2: GQL query layout

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. In the previous version of GQL, there is indicated that queries can be union-ed, intersected and the difference between two queries can be taken. Creation of a view:

```
CREATE QUERY <name> [<parameter list>] AS {
subquery
}
```

Listing 3: GQL view

Moreover, a view can also be utilized to return a graph to the user with the keyword CONSTRUCT.

Notice how the $ is used to denote a variable. Since the CALL option for joining views is not decided yet it will be left out, and for combining queries the union shall be used as described in [Mor].

Up to this point, there is no additional source that dives into the SQL/ PGQ. Hence the translation and derivation of that language are taken from [DFG+22].

```
CREATE QUERY viewNAME($input graphReturned)  AS {
FROM  $input
MATCH node-edge pattern
CONSTRUCT (node1)-[:viewNAME]-(node2)
}
}
```

Listing 4: GQL view graph returned

6 Discussion Conclusion

6.1 Discussion and Future Work

6.2 Conclusion

This is a test for generating the references

References

[ABD+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 2423–2436, 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+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 2246–2258, 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 377–392, New York, NY, USA, 2020. Association for Computing Machinery.

[FGG+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. 02 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.

[GSQa] GSQL. Ldbc snb : Business intelligence workload.

[GSQb] GSQL. Ldbc snb : Interactive intelligence workload.

[Har21] Harsh Vrajeshkumar Thakker. *On Supporting Interoperability between RDF and Property Graph Databases*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, May 2021.

[Mor] Olaf Morra. A semantics of gql a new query language for property graphs formalized.

[W3C] W3C. Knuth: Computers and typesetting.

7 Appendix

7.1 Business Intelligence queries

In the following subsections the business intelligence queries are translated or converted towards the Cypher style query.

7.1.1 Business Intelligence 1

```
CREATE QUERY messageCount_VIEW AS {
FROM message
MATCH (message:Message)
WHERE message.creationDate < $datetime
Return count(message) AS totalMessageCountInt}
```

5: message count

In 5 the message node is selected by the MATCH clause. After selection a verification is executed by the WHERE clause. The verification is based on whether the creation date of the message is lower than the provided date. At last, the count of the messages are returned as totalMessage Count.

```
CREATE  QUERY messagePrep_View AS (
From message
MATCH(message:message)
WHERE  creationDate < :datetime
    AND content IS NOT NULL
   SELECT extract(year from creationDate) AS messageYear
         , ParentMessageId IS NOT NULL AS isComment

Return ParentMessageId IS NOT NULL AS isComment,
       , CASE
             WHEN length <  40 THEN 0 -- short
             WHEN length <  80 THEN 1 -- one liner
             WHEN length < 160 THEN 2 -- tweet
             ELSE                   3 -- long
           END AS lengthCategory
         , length
)
```

6: message prep

In 6, the MATCH clause selects the message node. Although the query construction might seem similar to ??, it is quite different. After the match clause the where clause validates whether the original date of the message is lower than the actual date. Moreover, there is another verification carried out, which verifies to see whether there is content or not. At last, the id of the messages is returned with their respective length category.

The last query is the final query and uses the returned results from ?? and 6. Also here first the message node is selected and returns the results, which are grouped by the year, the length of the category, and the id of the parent message (variable: isComment).

The variables which are returned

7.1.2 Business Intelligence 2

In 11the message node is first selected through the optional match. After a selection of that node, a right path traversal is done towards the node tag along the path `HAS_TAG`.The where clause, which is present together with an

```
FROM messagePrep_VIEW, messageCount_VIEW
MATCH(message:message)
Return messageYear,
       isComment,
       lengthCategory,
       count(message) AS messageCount,
       sum(message.length)/toFloat (count(message)) AS averageMessageLength,
       sum(length) AS sumMessageLength,
       messageCount / toFloat(totalMessageCountInt) AS percentageOfMessages

 GROUP BY messageYear,
          isComment,
          lengthCategory,
 ORDER BY messageYear DESC,
          isComment ASC,
          lengthCategory ASC
```

7: final query

Variables	Meaning
messageYear	creation year of the message
isComment	id of the parent message
lengthCategory	length of the mesage categorized
messageCount	the count of how many messages based on the critera in the where clause of 7
averageMessageLength	average length of the message

Table 5: Returned variables and their meaning

```
Select count(message) AS totalMessageCount
FROM LDBC_SNB
    MATCH (message:Message)
WHERE message.creationDate < $datetime
```

8: message count

```
Select ParentMessageId IS NOT NULL AS isComment,
       , CASE
             WHEN length <  40 THEN 0 -- short
             WHEN length <  80 THEN 1 -- one liner
             WHEN length < 160 THEN 2 -- tweet
             ELSE                   3 -- long
           END AS lengthCategory
         , length
From LDBC_SNB
     MATCH(message:message)
WHERE  creationDate < :datetime
AND content IS NOT NULL
```

9: message prep

AND clause, verifies whether the message is newer than the provided date, but it should not be older than 100 days. At last, the count of the messages is returned under the variable name countWindow1.

In 12the match message node is selected again, and is somewhat similar to 11. The major difference is in the

```
SELECT messageYear, isComment, lengthCategory,count(message) AS messageCount,
sum(message.length)/toFloat (count(message)) AS averageMessageLength,
sum(length) AS sumMessageLength,
(messageCount / toFloat(totalMessageCountInt)) AS percentageOfMessages

FROM  messagePrep_VIEW, messageCount_VIEW
      MATCH(message:message)
GROUP BY messageYear,
          isComment,
          lengthCategory,
 ORDER BY messageYear DESC,
          isComment ASC,
          lengthCategory ASC
```

10: final query

```
CREATE QUERY TAG1_VIEW as (
FROM LDBC_SNB
OPTIONAL MATCH (message1:Message)-[:HAS_TAG]->(tag)
  WHERE $date <= message1.creationDate
    AND message1.creationDate < $date + duration({days: 100})
Return count(message1) AS countWindow1
)
```

11: tag view

```
CREATE QUERY TAG2_VIEW as (
FROM LDBC_SNB
OPTIONAL MATCH (message2:Message)-[:HAS_TAG]->(tag)
WHERE $date + duration({days: 100}) <= message2.creationDate
    AND message2.creationDate < $date + duration({days: 200})
Return
       count(DISTINCT CASE WHEN Message.creationDate >= :date + INTERVAL

       '100 days' THEN Message.id ELSE NULL END)
       AS countMonth2
)
```

12: tag2 view

WHERE clause.There the message must be 100 days newer than the provided creation date, but not an additionally 200 days newer.

In 13, which is the final query the view created in 11 and 12 are first selected. The MATCH CLAUSE selects the tag node and from there on a right path traversal is done towards the tagclass node. The traversal is done along the path named `HAS_TYPE`. At last, the variables are returned and ordered by the difference of the variable named diff (descending order)and the tag name in ascending order.

The returned variables are

7.1.3 Business Intelligence 3

The creation of a view is omitted in 17.In the MATCH clause there first is a left traversal from the forum node towards the person node. This traversal is done along the path `HAS_MODERATOR`, and a second traversal starts after that, along the path `IS_LOCATED_IN`. The second traversal is from the person node towards the city node, and another

```
FROM TAG1_VIEW,TAG2_VIEW,LDBC
MATCH (tag:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
RETURN tag.name
       countWindow1,
       countWindow2,
       abs(countWindow1 - countWindow2) AS diff
ORDER BY diff desc,
         tag.name ASC
LIMIT 100
```

13: final query

Variables name	Meaning
tag.name	name of the tag
countWindow1	count of the messages with a count lower than the creation date -100
countWindow2,	count of the messages with a count higher (100 days)than the creation dat, but lower than 200 days
diff	the absolute difference of countWindow1 and countWindow2

Table 6: Returned variables and their meaning

```
SELECT count(message1) AS countWindow1
FROM LDBC_SNB
     OPTIONAL MATCH (message1:Message)-[:HAS_TAG]->(tag)
WHERE $date <= message1.creationDate
AND message1.creationDate < $date + duration({days: 100})
```

14: tag view 1

```
SELECT count(DISTINCT CASE WHEN Message.creationDate >= :date + INTERVAL '100 days' THEN Message.id ELSE NULL END) AS countMonth2
FROM LDBC_SNB
     OPTIONAL MATCH (message2:Message)-[:HAS_TAG]->(tag)
WHERE $date + duration({days: 100}) <= message2.creationDate
AND message2.creationDate < $date + duration({days: 200})
```

15: tag view 2

```
SELECT  tag.name,countWindow1,countWindow2,abs(countWindow1 - countWindow2) AS diff
FROM TAG1_VIEW,TAG2_VIEW,LDBC
    MATCH (tag:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
ORDER BY diff desc,
         tag.name ASC
LIMIT 100
```

16: final query

left traversal is executed from the city node towards the country node. After those left traversals, there are also two right traversals and an undirected one. The first right traversal is from the forum node towards the post node and traverses through the path of `CONTAINER_OF`.Along the path of `HAS_TAG`, the second right traversal is done from the message node towards the tag node and the third towards the TagClass. `HAS_TYPE` is the path along which the third

```
FROM LDBC_SNB
MATCH
  (:Country {country: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-
  (person:Person)<-[:HAS_MODERATOR]-(f:FORUM)-[:CONTAINER_OF]->
  (post:Post)<-[:REPLY_OF]{0,...}-(message:Message)-[:HAS_TAG]->(:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
RETURN
f.id             AS "f.id"
f.title          AS "f.title"
f.creationDate    AS "f.creationDate"
f.ModeratorPersonId AS "person.id"
 count(DISTINCT MessageThread.MessageId) AS messageCount
GROUP BY f.id
        ,f.title
        ,f.creationDate
        ,f.ModeratorPersonId
 ORDER BY messageCount DESC,
          f.id
 LIMIT 20
```

17: final query

traversal is executed. Do note that there is another left traversal present from the message node to the post node that happens along the path `REPLY_OF`. The path traversal on the path is done at least zero or more times

The variables returned by the final query

Variables name	**Meaning**
f.id	id of the forum
,f.title	title of the forum
,f.creationDate	the date of the when the forum is created
,f.ModeratorPersonId	the id of the person who is the moderator

Table 7: Returned variables and their meaning

```
SELECT Forum.id AS "forum.id", Forum.title  AS "forum.title",
     Forum.creationDate   AS "forum.creationDate",
      Forum.ModeratorPersonId AS "person.id",
      count(DISTINCT MessageThread.MessageId) AS messageCount
FROM LDBC_SNB
      MATCH (:Country {country: $country})<-[:IS_PART_OF]-(:City)
             <-[:IS_LOCATED_IN]-(person:Person)<-[:HAS_MODERATOR]-(f:Forum)
             -[:CONTAINER_OF]->(post:Post)<-[:REPLY_OF]{0,...}
             -(message:Message)-[:HAS_TAG]->(:Tag)

             -[:HAS_TYPE] ->(:TagClass {name:$tagClass})
GROUP BY f.id
        ,f.title
        ,f.creationDate
        ,f.ModeratorPersonId
 ORDER BY messageCount DESC,
          Forum.id
 LIMIT 20
```

18: final query

7.1.4 Business Intelligence 4

```
Create Query as ForumMembershipPerCountry_VIEW (
FROM LDBC_SNB
MATCH (country:Country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(person:Person)<-[:HAS_MEMBER]-(forum:Forum)
Return Forum.id AS ForumId
      , count(Person.id) AS numberOfMembers
      , Country.id AS CountryId
      ,  DISTINCT forum AS topForum1
 GROUP BY Country.Id, Forum.Id
)
```

19: Membership per forum per Country

In 22 the query first selects the city node and from there on a left path traversal is done towards the country node along the path of `IS_PART_OF`. A second left path traversal is carried out towards the city node, which comes from the forum node through the path named `HAS_MEMBER`.

```
Create Query as  Top100_Popular_Forums_VIEW(
FROM ForumMembershipPerCountry_VIEW,LDBC_SNB
MATCH (person:Person)<-[:HAS_MEMBER]-(topForum2:Forum)
OPTIONAL MATCH (topForum1)
-[:CONTAINER_OF]->(post:Post)
<-[:REPLY_OF]{0,...}-(message:Message)-[:HAS_CREATOR]->(person)
WHERE topForum2 IN topForums
RETURN DISTINCT ForumId AS id
        ,max(numberOfMembers) AS maxNumberOfMembers
        person,
        message,
        topForum2
  GROUP BY ForumId

  ORDER BY maxNumberOfMembers DESC
          ,ForumId
  LIMIT 100
)
```

20: Top 100Popular Forums

The `ForumMembershipPerCountry_VIEW` query has two MATCH clauses. In the first match clause, there is a left path traversal from the forum node towards the person node. The path traversal is done along the path named `HAS_MEMBER`. The OPTIONAL MATCH has two right path traversals and one left path traversal. The first right path traversal is from topforum 1 node or the results retrieved from 22 and goes towards the post node, through the path named `CONTAINER_OF`. The second traversal is from the message node towards the person node and travels along the path of `HAS_CREATOR`.

The last query is the final one and selects all the results from 22 and 4-2, and returns the variables, which are described in table 8

personId	the id of the person
personFirstName	the first name
personLastName	last name
personCreationDate	the date at which the account was created
messageCount	the amount of messages sent by a user in a forum

Table 8: Returned variables and their meaning

```
FROM Top100_Popular_Forums_VIEW,ForumMembershipPerCountry_VIEW
MATCH ()
 RETURN person.id AS personId,
  person.firstName AS personFirstName,
  person.lastName AS personLastName,
  person.creationDate AS personCreationDate,
  count(DISTINCT message) AS messageCount
ORDER BY
  messageCount DESC,
  person.id ASC
LIMIT 100
```

21: Final Query

```
//ForumMembershipPerCountry_VIEW

SELECT Forum.id AS ForumId,
        count(Person.id) AS numberOfMember,Country.id AS CountryId,
        DISTINCT forum AS topForum1
FROM LDBC_SNB
     MATCH (country:Country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]
     -(person:Person)<-[:HAS_MEMBER]-(forum:Forum)
 GROUP BY Country.Id
           ,Forum.Id
```

22: ForumMembership Per Country

```
//Top100_Popular_Forums_VIEW

SELECT DISTINCT ForumId AS id,max(numberOfMembers) AS maxNumberOfMembers
                ,person, message, topForum2
FROM ForumMembershipPerCountry_VIEW,LDBC_SNB
     MATCH (person:Person)<-[:HAS_MEMBER]-(topForum2:Forum)
     OPTIONAL MATCH (topForum1)-[:CONTAINER_OF]->(post:Post)
                     <-[:REPLY_OF]{0,...}
                     -(message:Message)-[:HAS_CREATOR]->(person)
WHERE topForum2 IN topForums
  GROUP BY ForumId

  ORDER BY maxNumberOfMembers DESC
           ,ForumId
  LIMIT 100
```

23: Top 100 Popular Forums VIEW

7.1.5 Business Intelligence 5

In 26 the query has two optional MATCH clauses and one MATCH clause. In the MATCH clause, there is a left, and right. the left path traversal starts from the message node towards the tag node, and traverses through the path with label named HAS_TAG, while the right path traversal happens through the HAS_CREATOR label. The label comes from the message node and goes toward the person node At last, the query returns the variables described in table 9

7.1.6 Business Intelligence 6

In 31 the left traversal goes through the label HAS_TAGand from the message node towards the tag node. Also,

```
Select person.id AS personId,person.firstName AS personFirstName
     ,person.lastName AS personLastName, person.creationDate AS personCreationDate
     ,count(DISTINCT message) AS messageCount
FROM Top100_Popular_Forums_VIEW,ForumMembershipPerCountry_VIEW
     MATCH ()
ORDER BY
  messageCount DESC,
  person.id ASC
LIMIT 100
```

24: Top 100 Popular Forums VIEW

```
FROM LDBC_SNB
MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(p:Person)
OPTIONAL MATCH (message)<-[likes:LIKES]-(:Person)
OPTIONAL MATCH (message)<-[:REPLY_OF]-(reply:Comment)
REUTRN CreatorPerson.id AS person.id
     , count(DISTINCT Comment.Id)  AS replyCount
     , count(DISTINCT Person_likes_Message.MessageId||' '||Person_likes_Message.PersonId) AS likeCount
     , count(DISTINCT Message.Id)  AS messageCount
     ,1*messageCount + 2*replyCount + 10*likeCount AS score

 GROUP BY CreatorPerson.id
 ORDER BY score DESC, CreatorPersonId
 LIMIT 100
```

25: final query

person.id	the id of the person
replyCount	the amount of times there is replied on a message
likeCount	count of likes
messageCount	the amount of distinct messages
score	the score of the message dependant on the count of likes, the amount of times a reply is done and the messa

Table 9: Returned variables and their meaning

```
SELECT CreatorPerson.id AS person.id
     , count(DISTINCT Comment.Id)  AS replyCount
     , count(DISTINCT Person_likes_Message.MessageId||' '||Person_likes_Message.PersonId) AS likeCount
     , count(DISTINCT Message.Id)  AS messageCount
     ,1*messageCount + 2*replyCount + 10*likeCount AS score
FROM LDBC_SNB
      MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(p:Person)
      OPTIONAL MATCH (message)<-[likes:LIKES]-(:Person)
      OPTIONAL MATCH (message)<-[:REPLY_OF]-(reply:Comment)
 GROUP BY CreatorPerson.id
 ORDER BY score DESC, CreatorPersonId
 LIMIT 100
```

26: final query

```
CREATE QUERY detail_VIEW as (
FROM LDBC_SNB
MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(person:Person)
OPTIONAL MATCH [:REPLY_OF]-(reply:Comment)->(message)<-[likes:LIKES]-(:Person)
REUTRN CreatorPerson.id AS CreatorPersonId
     , count(DISTINCT Comment.Id)  AS replyCount
     , count(DISTINCT Person_likes_Message.MessageId||' '||Person_likes_Message.PersonId) AS likeCount
     , count(DISTINCT Message.Id)  AS messageCount
     , NULL as score
GROUP BY CreatorPerson.id
)
```

27: detail view

from the message node, there is another right path traversal towards the person node, which happens along the path HAS_CREATOR. Both traversals happen within the MATCH CLAUSE. The other match clause, which is an OPTIONAL MATCH has a left path traversal from the person node towards the message node. In there

```
Create Query poster_w_liker_VIEW AS(
FROM detail_VIEW,LDBC_SNB
MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message1:Message)-[:HAS_CREATOR]->(person1:Person)
OPTIONAL MATCH (message1)<-[:LIKES]-(person2:Person)
RETURN    DISTINCT   m1.CreatorPersonId AS posterPersonid
         ,l2.PersonId AS likerPersonid
)
```

28: poster liker view

The poster_w_liker_VIEW has a left traversal through the path HAS_TAG, which comes from the message node towards the tag node. From the message node towards the person node, there is a right path traversal that travels through the HAS_CREATOR node.

```
CREATE QUERY popularity_score_VIEW AS (
FROM LDBC_SNB,poster_w_liker_VIEW
OPTIONAL MATCH (person2)<-[:HAS_CREATOR]-(message2:Message)<-[like:LIKES]-(person3:Person)
RETURN  CreatorPersonId AS PersonId
        ,count(*) AS popularityScore
GROUP BY m3.CreatorPersonId
)
```

29: Popularity score

The popularity_score view has two left path traversals. The first starts from the person node and travels through the LIKES label towards the message node (from 31). From that message node, the second traversal is done from the message node towards the person node of 34. That traversal is done through the HAS_CREATOR label.

The final query also has a MATCH CLAUSE. This clause does a left path traversal from the person node towards the likerPersonid, and travels along the path named POPULARITY_SCORE. The variables returned from the final query are described in table 7.1.7

7.1.7 Business Intelligence 7

The query above traverses from the message node towards the tag node. This traversal happens through the HAS_TAG label. A second traversal is present in the MATCH clause, which consists of a left path traversal and a

```
/*Final Query*/
FROM poster_w_liker_VIEW,popularity_score_VIEW,Query poster_w_liker_VIEW ,detail_VIEW
MATCH (pl: likerPersonid) <- [:POPULARITY_SCORE]-(ps:Person)
RETURN  pl.posterPersonid AS "person1.id"
     , sum(coalesce(ps.popularityScore, 0)) AS authorityScore
 GROUP BY pl.posterPersonid
 ORDER BY authorityScore DESC, pl.posterPersonid ASC
 LIMIT 100
;
```

30: final query

Variables	Meaning
person1.id	the id of the person
authorityScore	sum of the popularity score retrieved from the traversal in the last queryn

Table 10: Returned variables and their meaning

```
// detail_VIEW
SELECT CreatorPerson.id AS CreatorPersonId
     , count(DISTINCT Comment.Id)  AS replyCount
     , count(DISTINCT Person_likes_Message.MessageId||' '||Person_likes_Message.PersonId) AS likeCount
     , count(DISTINCT Message.Id)  AS messageCount
     , NULL as score
FROM LDBC_SNB
    MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(person:Person)
    OPTIONAL MATCH (message)<-[likes:LIKES]-(:Person)
    OPTIONAL MATCH (message)<-[:REPLY_OF]-(reply:Comment)
GROUP BY CreatorPerson.id
```

31: detail view

```
//poster_w_liker_VIEW
SELECT DISTINCT  m1.CreatorPersonId AS posterPersonid
        ,l2.PersonId AS likerPersonid
FROM detail_VIEW,LDBC_SNB
     MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message1:Message)-[:HAS_CREATOR]->(person1:Person)
     OPTIONAL MATCH (message1)<-[:LIKES]-(person2:Person)
```

32: poster liker view

```
//popularity_score_VIEW
SELECT  CreatorPersonId AS PersonId, count(*) AS popularityScore
FROM Message m3
     OPTIONAL MATCH (person2)<-[:HAS_CREATOR]-(message2:Message)<-[like:LIKES]-(person3:Person)
GROUP BY m3.CreatorPersonId
```

33: Popularity score

right path traversal, unlike the previous traversal that is only a left path traversal. The first right path traversal is

```
//final query
SELECT  pl.posterPersonid AS "person1.id"
       ,sum(COALESCE(ps.popularityScore, 0)) AS authorityScore
FROM poster_w_liker_VIEW
MATCH (pl: likerPersonid) <- [:POPULARITY_SCORE] (ps:Person)

 GROUP BY pl.posterPersonid
 ORDER BY authorityScore DESC
         ,pl.posterPersonid ASC
LIMIT 100
```

34: final query

```
/*Translation Bi7 query*/
FROM LDBC_SNB
MATCH
  (tag:Tag {name: $tag})<-[:HAS_TAG]-(m:Message),
  (m)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)
    WHERE NOT (c)-[:HAS_TAG]->(tag)

RETURN RelatedTag.name AS "relatedTag.name"
       , count(*) AS count
 GROUP BY RelatedTag.name
 ORDER BY count DESC
          ,RelatedTag.name
 LIMIT 100
```

35: final query

from the comment node and travels through the HAS_TAG path towards the Tag node, while the left path traversal goes from the comment node towards the message node.REPLY_OF is the path through which the left path traversal happens.

```
SELECT RelatedTag.name AS "relatedTag.name",count(*) AS count
FROM LDBC_SNB
     MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(m:Message),
           (m)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)

            WHERE NOT (c)-[:HAS_TAG]->(tag)

GROUP BY RelatedTag.name
ORDER BY count DESC
          ,RelatedTag.name
LIMIT 100
```

36: final query

The 39 is composed with a WHERE EXIST CLAUSE. In there the query starts with a left path traversal from

Variables	**Meaning**
relatedTag.name	the name of the tag
count	count of the messages related to the tag ID

```
/*Translation Bi8 query*/

CREATE Query Person_VIEW as (
FROM  Person
MATCH (tag:Tag {name: $tag})
// score
MATCH (tag)<-[interest:HAS_INTEREST]-(person:Person)
tag, collect(person) AS interestedPersons
WHERE EXIST{
    MATCH (tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(person:Person)
}
RETURN DISTINCT tag
       ,DISTINCT(interestedPersons + COLLECT (person) )AS persons
)
```

37: Person view

the message node towards the tag node, traversing through the path named `HAS_TAG`. The other path `HAS_CREATOR` has a right traversal from the message node towards the person node. The inner clause is a condition, therefore the outer MATCH clause first gets executed. In that clause, the person node is selected and from there on a traversal is done towards the tag node through the `HAS_INTEREST` label.

7.1.8 Business Intelligence 8

```
FROM Person_VIEW
 MATCH (person)-[:KNOWS]-(friend)
// We need to use a redundant computation due to the lack of composable graph queries in the currently supported Cypher version.
// This might change in the future with new Cypher versions and GQL.
RETURN
 tag,
  person,
  100 * size([(tag)<-[interest:HAS_INTEREST]-(person) | interest]) + size([(tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(p
  AS score
  ,sum(score) AS friendScore

ORDER BY
  score + friendsScore DESC,
  person.id ASC
LIMIT 100
```

38: Final query

The last query retrieves the results from the `person_VIEW`. This however, might be redundant ,and could be constructed in one query. But the current version of GQL described in the available resources does not mention composable graph queries. As a result, a second query is constructed as the final query. In it we have a traversal from the friend node towards the person node. This traversal is a LUR and happens through the KNOWS path. As a final step the variables described in table 11 are returned.

Variables
tag,
100 * size([(tag)<-[interest:HAS_INTEREST]-(person) | interest]) + size([(tag)<-[:HAS_TAG]-(message:Message)-[:HAS_C

Table 11: Returned variables and their meaning

```
SELECT DISTINCT tag
       ,DISTINCT(interestedPersons + COLLECT (person) )AS persons
FROM  LDBC_SNB
      MATCH (tag:Tag {name: $tag}),(tag)<-[interest:HAS_INTEREST]-(person:Person)
             tag, collect(person) AS interestedPersons
WHERE EXIST{
    MATCH (tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(person:Person)
}
```

39: Person view

```
//final query
SELECT tag, person,
  100 * size([(tag)<-[interest:HAS_INTEREST]-(person) | interest]) + size([(tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(p
  AS score
  ,sum(score) AS friendScore
FROM LDBC_SNB,Person_VIEW
     MATCH (person)-[:KNOWS]-(friend)
ORDER BY
  score + friendsScore DESC,
  person.id ASC
LIMIT 100
```

40: Final query

7.1.9 Business Intelligence 9

```
FROM LDBC_SNB
MATCH (person:Person)<-[:HAS_CREATOR]-(post:Post)<-[:REPLY_OF]{0,1}-(reply:Message)
WHERE  post.creationDate >= $startDate
  AND  post.creationDate <= $endDate
  AND reply.creationDate >= $startDate
  AND reply.creationDate <= $endDate
RETURN person.id
      ,person.firstName
      ,person.lastName
      ,count(DISTINCT post) AS threadCount
      ,count(DISTINCT reply) AS messageCount
ORDER BY
  messageCount DESC,
  person.id ASC
LIMIT 100
 LIMIT 100
```

41: final query

The query starts of by having two left path traversals. The first is from the message node, and traverses to the post node. During traversal between 0 and 1 path are traversed.The traversal is executed along the path named `REPLY_OF`.Second traversal ends at the person node,which starts from the post node, and travels through the path `HAS_CREATOR`. At last variables are returned which are described in table 12

7.1.10 Business Intelligence 11

Variables	Meaning
person.id	id of the person
person.firstName	first name of the person
person.lastName	last name of the person
threadCount	amount of posts by that individual
messageCount	amount of replies by that same individual

Table 12: Returned variables and their meaning

```
 RelatedTag.name AS "relatedTag.name",count(*) AS count
FROM LDBC_SNB
     MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(m:Message),
           (m)<-[:REPLY_OF]-(c:Comment)-[:HAS_TAG]->(rTag:Tag)

            WHERE NOT (c)-[:HAS_TAG]->(tag)

GROUP BY RelatedTag.name
ORDER BY count DESC
          ,RelatedTag.name
LIMIT 100
```

42: final query

```
FROM LDBC_SNB
MATCH (country:Country {name: $country}),
 (person1:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
 (person2:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
 (person3:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
 (person1)-[k1:KNOWS]-(person2)-[k2:KNOWS]-(person3)-[k3:KNOWS]-(a)
WHERE a.id < b.id
  AND b.id < c.id
  AND $startDate <= k1.creationDate <= k1.creationDate
  AND $startDate <= k2.creationDate
  AND $startDate <= k3.creationDate
RETURN count(*) AS count
       ,person1
       ,person2
       ,person3
```

43: final query

The query has in the match clause six right path traversals. The first traversal happens from the person node towards the city node and travels through the `IS_LOCATED_IN` path. From the city node, the second traversal happens toward the country node through the label `IS_PART_OF`. The other right path traversals follow the same routine. The only difference is that there are three different person node traversals towards the country node. In the end a LUR is executed to see whether the three persons know each other, and the query returns the variables described in the table 13

Variables	Meaning
person1	results retrieved for the person 1 node
person2	results retrieved for the person 2 node
person3	results retrieved for ther person 3 node

Table 13: Returned variables and their meaning

```
SELECT count(*) AS count
       ,person1
       ,person2
       ,person3
FROM LDBC_SNB
     MATCH (country:Country {name: $country}),
           (person1:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
           (person2:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
           (person3:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
           (person1)-[k1:KNOWS]-(person2)-[k2:KNOWS]-(person3)-[k3:KNOWS]-(a)
WHERE a.id < b.id
  AND b.id < c.id
  AND $startDate <= k1.creationDate <= k1.creationDate
  AND $startDate <= k2.creationDate
  AND $startDate <= k3.creationDate
```

44: final query

7.1.11 Business Intelligence 12

```
FROM  LDBC_SNB
OPTIONAL MATCH (person: person)<-[:HAS_CREATOR]-(message:Message)-[:REPLY_OF]{0,..}->(post:Post)
WHERE message.content IS NOT NULL
  AND message.length < 20
  AND message.creationDate > \$startDate
  AND post.language IN   ['ar', 'hu']
RETURN
  messageCount,
  count(person) AS personCount
ORDER BY
  personCount DESC,
  messageCount DESC
```

45: Final Query

From the person node, there is an incoming left traversal from the message node, which travels through the `HAS_CREATOR` path. Another traversal is present, which travels toward the post node, of which at least 0 paths are traversed through the `REPLY_OF` path.The WHERE clause described the conditions for the retrieved results.The query returns the variable messageCount, which indicates the amount of messages created within the specified conditions

7.1.12 Business Intelligence 13

From the query above we have the person node and from there onwards there are only left path traversals. The first one ends at the city node, which travels through the path named `IS_PART_OF`.After traversal of that path a second path is followed named `IS_LOCATED_IN` and ends at the country node. Furthermore, the query has a WHERE condition. In the condition, a check is executed to see whether the date of the created profile is lower than the provided end date.

In 48 the OPTIONAL MATCH clause has a left path traversal from the message node towards the Person node through the path `HAS_CREATOR`. Also in this subquery, the WHERE condition verifies whether the created profile date is lower than the end date.

The query above has an OPTIONAL MATCH, in which a left path traversal is done from the person node towards the message node through the `HAS_CREATOR` path towards another person node. Do note that this WHERE clause has another type of condition. In there a check is done to see whether those two results retrieved from the node are identical. At last, another optional MATCH is present which retrieves the results from 47 and 48. The query returns the variables described in table 7.1.12

```
SELECT messageCount,
       count(person) AS personCount
 FROM  LDBC_SNB
       OPTIONAL MATCH (person: person)<-[:HAS_CREATOR]-(message:Message)
                     -[:REPLY_OF]{0,..}->(post:Post)
 WHERE message.content IS NOT NULL
   AND message.length < 20
   AND message.creationDate > $startDate
   AND post.language IN   ['ar', 'hu']
 ORDER BY
   personCount DESC,
   messageCount DESC
```

46: final query

```
CREATE QUERY Located_VIEW as (
FROM LDBC_SNB
MATCH (country:Country {name: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(:Person)
WHERE person.creationDate < $endDate
RETURN country
       ,person
_)
```

47: Location Retrieval

```
CREATE QUERY HAS_CREATOR_VIEW as (
OPTIONAL MATCH (person)<-[:HAS_CREATOR]-(message:Message)
WHERE message.creationDate < $endDate
    AND  messageCount / months < 1
RETURN  country
       ,person
       ,count(message) AS messageCount

)
```

48: Creator of the message

Variables	**Meaning**
person.id	the id of person
personLikeCount	amount a person liked a comment or message
totalLikeCount	the total amount of messages/comments liked by a person

Table 14: Variables and their meaining

7.1.13 Business Intelligence 14

In the **??** the MATCH clause consists of three independent clauses. The first clause executes a traversal from the person node towards the city node through the path of the `IS_LOCATED_IN`. From the city node, there is another left traversal, which goes to the country node through the path `IS_PART_OF`. The last clause verifies whether the two people know each other.

In 55 the MATCH clause has a left path traversal as well as a right one. The right one starts from the message node and moves towards the person node, through the label 55. Towards the message node, there is another incoming

```
CREATE QUERY LikerPerson_VIEW as (
OPTIONAL MATCH
  (person)<-[:HAS_CREATOR]-(message:Message)<-[:LIKES]-(likerperson:Person)
WHERE likerperson IN persons
RETURN country,
       COLLECT (person) AS persons
         12 * ($endDate.year  - person.creationDate.year )
     + ($endDate.month - person.creationDate.month)
     + 1 AS months
     ,messageCount
)
```

49: Person likes Person message

```
FROM LDBC_SNB,LikerPerson_VIEW,HAS_CREATOR_VIEW,Located_VIEW
OPTIONAL MATCH
  (person)<-[:HAS_CREATOR]-(message:Message)<-[:LIKES]-(likerPerson:Person)
WHERE likerPerson.creationDate < $endDate

RETURN person.id
       ,count(likerperson) AS personLikeCount
       ,totalLikeCount
  CASE totalLikeCount
    WHEN 0 THEN 0.0
    ELSE personLikeCount / toFloat(totalLikeCount)
  END AS personScore
ORDER BY
  personScore DESC,
  person.id ASC
LIMIT 100
```

50: final query

```
SELECT country
       ,zombie
       ,count(message) AS messageCount
FROM LDBC_SNB
MATCH (country:Country {name: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(zombie:Person)
WHERE zombie.creationDate < $endDate
RETURN country, zombie
OPTIONAL MATCH (zombie)<-[:HAS_CREATOR]-(message:Message)
WHERE message.creationDate < $endDate
    AND  messageCount / months < 1
```

51: final query

right path traversal from the comment node, which comes through the `REPLY_OF` path. From the coming node, there is a left path traversal from the comment node towards the person node and travels through the path `HAS_CREATOR`.

The contents of the OPTIONAL MATCH are similar to that of CASE 1 in 56. However, the return statements differ. The difference lies in the score variable.

In 57 and 58 the traversal is almost similar. The main difference is that in 58 there is a right path traversal from the person1 node towards the message node through the path named `LIKES`, and the second right path traversal is from the message node through the path `HAS_CREATOR` towards the person2 node, whereas in 58 the traversal is a

```
SELECT country,
       collect(zombie) AS zombies
         12 * ($endDate.year  - zombie.creationDate.year )
     + ($endDate.month - zombie.creationDate.month)
     + 1 AS months
     ,messageCount
FROM  ,LDBC_SNB
OPTIONAL MATCH
  (zombie)<-[:HAS_CREATOR]-(message:Message)<-[:LIKES]-(likerZombie:Person)
WHERE likerZombie IN zombies
```

52: final query

```
SELECT zombie.id
       ,count(likerZombie) AS zombieLikeCount
       ,zombieLikeCount,
       ,totalLikeCount,
    CASE totalLikeCount
    WHEN 0 THEN 0.0
    ELSE zombieLikeCount / toFloat(totalLikeCount)
    END AS zombieScore
FROM ,LDBC
OPTIONAL MATCH (zombie)<-[:HAS_CREATOR]-(message:Message)<-[:LIKES]-(likerPerson:Person)
WHERE likerPerson.creationDate < $endDate
ORDER BY
  zombieScore DESC,
  zombie.id ASC
LIMIT 100
```

53: final query

```
/*Translate Bi14 query*/
CREATE QUERY KNOW_VIEW AS (
FROM LBC_SNB
MATCH
  (country1:Country {name: $country1})<-[:IS_PART_OF]-(city1:City)<-[:IS_LOCATED_IN]-(person1:Person),
  (country2:Country {name: $country2})<-[:IS_PART_OF]-(city2:City)<-[:IS_LOCATED_IN]-(person2:Person),
  (person1)-[:KNOWS]-(person2)
Return person1
       ,person2
       ,city1
       ,0 AS score
)
```

54: Know view

left path, and the labels are exchanged. The query returns the following variables described in table 15

7.1.14 Business Intelligence 18

In 64 the query has two MATCH clauses. In the first MATCH clause, the query first traverses from the person node towards the tag node, in a left manner through the path named `HAS_INTEREST`. From the person node, there is a LUR towards itself that travels through the label `KNOWS`. From the person node then there is a right traversal

```
//case 1
FROM LDBC_SNB
OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(c:Comment)-[:REPLY_OF]->(:Message)-[:HAS_CREATOR]->(person2)
Return DISTINCT person1
               ,person2
               ,city1
               ,score + (CASE c WHEN null THEN 0 ELSE  4 END) AS score
```

55: Case 1

```
// case 2// Ommit FROM, it requires person from the first tag
FROM LDBC_SNB
OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(m:Message)<-[:REPLY_OF]-(:Comment)-[:HAS_CREATOR]->(person2)
RETURN DISTINCT person1
                ,person2
                ,city1
                ,score + (CASE m WHEN null THEN 0 ELSE  1 END) AS score
```

56: Case 2

```
// case 3
FROM LDBC_SNB
OPTIONAL MATCH (person1)-[:LIKES]->(m:Message)-[:HAS_CREATOR]->(person2)
RETURN  DISTINCT person1
                ,person2
                ,city1
                ,score + (CASE m WHEN null THEN 0 ELSE 10 END) AS score
```

57: Case 3

```
// case 4
FROM LDBC_SNB
OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(m:Message)<-[:LIKES]-(person2)
RETURN DISTINCT person1.id
                ,person2.id
                ,city1.name
                ,score + (CASE m WHEN null THEN 0 ELSE  1 END) AS score DESC
ORDER BY
  top.score DESC,
  top.person1.id ASC,
  top.person2.id ASC
```

58: Final Query

from the person node towards the tag node. That traversal is executed through the path named `HAS_INTEREST`. The second OPTIONAL MATCH does another traversal from the person node towards the person node itself through the label `KNOWS`. This is done to see whether person1 is a mutual friend of person 2. At last, the following variables are returned as described in table 16

Variables	Meaning
person1.id	id of person retrieved from node person1
person2.id	id of person retrieved from node person 2
city1	city the person lives in

Table 15: Returned variables and their meaning

```
SELECT  person1
       ,person2
       ,city1
       ,0 AS score

FROM LBC_SNB
     MATCH (country1:Country {name: $country1})<-[:IS_PART_OF]-(city1:City)<-[:IS_LOCATED_IN]-(person1:Person),
           (country2:Country {name: $country2})<-[:IS_PART_OF]-(city2:City)<-[:IS_LOCATED_IN]-(person2:Person),
           (person1)-[:KNOWS]-(person2)
```

59: final query

```
SELECT DISTINCT person1
               ,person2
               ,city1
               ,score + (CASE c WHEN null THEN 0 ELSE  4 END) AS score
FROM LBC_SNB
     OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(c:Comment)-[:REPLY_OF]->(:Message)-[:HAS_CREATOR]->(person2)
```

60: Mutual Friend

```
SELECT DISTINCT person1,person2,city1,score + (CASE m WHEN null THEN 0 ELSE  1 END) AS score
FROM LBC_SNB
      OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(m:Message)<-[:REPLY_OF]-(:Comment)-[:HAS_CREATOR]->(person2)
```

61: Creator of the message

```
SELECT  DISTINCT person1 ,person2 ,city1,score + (CASE m WHEN null THEN 0 ELSE 10 END) AS score
FROM LBC_SNB
     OPTIONAL MATCH (person1)-[:LIKES]->(m:Message)-[:HAS_CREATOR]->(person2)
```

62: extended message creation results

Variables	Meaning
idPerson1	the id of person 1
idPerson2	id of person 2
mutualFriendCount	the amount of mutual friends

Table 16: Returned variables and their meaning

```
SELECT DISTINCT person1.id,person2.id,city1.name,score + (CASE m WHEN null THEN 0 ELSE  1 END) AS score DESC
FROM LBC_SNB
     OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(m:Message)<-[:LIKES]-(person2)
ORDER BY
  top.score DESC,
  top.person1.id ASC,
  top.person2.id ASC
```

63: final query

```
FROM  LDBC_SNB
MATCH (tag:Tag {name: $tag})<-[:HAS_INTEREST]-(person1:Person)-[:KNOWS]-(commonFriend:Person)-[:HAS_INTEREST]->(tag)
/*Reason for an optional match is the left join in sql*/
OPTIONAL MATCH (commonFriend:Person)-[:KNOWS]-(person2:Person)
WHERE person1 <> person2
  AND NOT (person1)-[:KNOWS]-(person2)
RETURN person1.id AS person1Id
      ,person2.id AS person2Id
      ,count(DISTINCT mutualFriend) AS mutualFriendCount
ORDER BY mutualFriendCount DESC, person1Id ASC, person2Id ASC
LIMIT 20
```

64: final query

```
SELECT person1.id AS person1Id,person2.id AS person2Id,count(DISTINCT mutualFriend) AS mutualFriendCount
FROM  LDBC_SNB
MATCH (tag:Tag {name: $tag})<-[:HAS_INTEREST]-(person1:Person)-[:KNOWS]-(mutualFriend:Person)-[:HAS_INTEREST]->(tag)
OPTIONAL MATCH (mutualFriend:Person)
-[:KNOWS]-(person2:Person)
WHERE person1 <> person2
  AND NOT (person1)-[:KNOWS]-(person2)
ORDER BY mutualFriendCount DESC, person1Id ASC, person2Id ASC
LIMIT 20
```

65: final query

```
SELECT person1.id AS person1Id,person2.id AS person2Id,count(DISTINCT mutualFriend) AS mutualFriendCount
FROM  LDBC_SNB
MATCH (tag:Tag {name: $tag})<-[:HAS_INTEREST]-(person1:Person)-[:KNOWS]-(mutualFriend:Person)-[:HAS_INTEREST]->(tag)
OPTIONAL MATCH (mutualFriend:Person)
-[:KNOWS]-(person2:Person)
WHERE person1 <> person2
  AND NOT (person1)-[:KNOWS]-(person2)
ORDER BY mutualFriendCount DESC, person1Id ASC, person2Id ASC
LIMIT 20
```

66: final query

7.2 Interactive Queries

In the following paragraphs the interactive queries are shown along with the variables meaning, return size and description of each query.

7.2.1 Interactive Complex 1

Since GQL does not yet support accumulators the query is constructed by first creating small views that return the results. Each consequent view is a follow-up of the previous.In `Query_friend_VIEW1` the query first accumulates all the person's first names and last names, the friends of each other. Then the person's ID and that friend are returned In the second view `friend_VIEW2` the query first retrieves the shorted path between person and friend, of which the

```
CREATE QUERY friend_VIEW1  AS (
FROM LDBC_SNB
MATCH (p:Person {id: $personId}) ~[:friend] ~ (Person {firstName: $firstName})
RETURN p
      ,friend
)
```

path can be between 1 and 3 , indicated by 1,3. After that the minimal length is returned from that traversed path and the friend as second result.

```
CREATE QUERY  friend_VIEW2 AS (
FROM LDBC_SNB, friend_VIEW1
MATCH path = shortestPath((p)-[:KNOWS]-(f:friend)) {1,3}
WHERE p.id == friend_VIEW1.p.id
Return min(length(path)) AS distance
       , f
ORDER BY
    distance ASC,
    f.lastName ASC,
    toInteger(f.id) ASC
LIMIT 20
```

In the following query the use of `friend_VIEW2` is used, since the distance result is required from the previous views. At first the query looks in which city each friend is located. In the second match (optional match), the the city in which each friend or person studies is retrieved. The returned results are the uni name provided a value of T if the university name is missing or else the name, class year city, and the distance value.

```
CREATE QUERY friend_VIEW3 as (
MATCH (f:friend)-[:IS_LOCATED_IN]
       ->(friendCity:City)
OPTIONAL MATCH (friend)-[studyAt:STUDY_AT]
            ->(uni:University)-[:IS_LOCATED_IN]->(uniCity:City)
CASE uni.name
        WHEN null T
        THEN null
        ELSE [uni.name, studyAt.classYear, uniCity.name]
 END ) AS
WHERE f.id == friend_VIEW1.f.id
 Return  unis
        ,friendCity
        ,distance
)
```

The query below is the last query without a view. I In there the company information at where that friend work is obtained by first traversing the path between friend and works at the company and afterwards the path between company and the country at which the company is located.After path traversal he company name is returned (if not null), university information of `friend_VIEW3`, and the distance. Furthermore all remaining results of the views are returned.

```
FROM friend_VIEW1,friend_VIEW2,friend_VIEW3
OPTIONAL MATCH (f:friend)-[workAt:WORK_AT]->(company:Company)-[:IS_LOCATED_IN]
           ->(companyCountry:Country)
CASE company.name
        WHEN null THEN null
        ELSE [company.name, workAt.workFrom, companyCountry.name]
END ) AS companies
WHERE f.id = friend_VIEW1.id
Return
        p.id AS friendId,
        f.lastName AS friendLastName,
        distance   AS distanceFromPerson,
        p.birthday AS friendBirthday,
        p.creationDate AS friendCreationDate,
        p.gender AS friendGender,
        p.browserUsed AS friendBrowserUsed,
        p.locationIp AS friendLocationIp,
        p.email AS friendEmails,
        p.speaks AS friendSpeaks,
        friendCityName AS friendCityName,
        companies AS friendCompanies,
        universities as friendUniversities
```

```
SELECT p,friend
FROM LDBC_SNB
     MATCH (p:Person {id: $personId}) ~[:friend] ~ (Person {firstName: $firstName})
```

```
//friend_VIEW2
SELECT min(length(path)) AS distance,f
FROM LDBC_SNB, friend_VIEW1
     MATCH path = shortestPath((p)-[:KNOWS]-(f:friend)) {1,3}
     WHERE p.id == friend_VIEW1.p.id
ORDER BY
    distance ASC,
    f.lastName ASC,
    toInteger(f.id) ASC
LIMIT 20
```

7.2.2 Interactive Complex 2

```
FROM LDBC_SNB
MATCH (P:Person {id: $personId })-[:KNOWS]-(f:Person)<-[:HAS_CREATOR]-(m:Message)
WHERE message.creationDate <= $maxDate
CASE :
    m.content <> ""
    THEN
            result = coalesce(m.content,m.imageFile)
```

```
SELECT  unis,friendCity,distance
MATCH (f:friend)-[:IS_LOCATED_IN]->(friendCity:City)
OPTIONAL MATCH (friend)-[studyAt:STUDY_AT]->(uni:University)-[:IS_LOCATED_IN]->(uniCity:City)
CASE uni.name
        WHEN null T
        THEN null
        ELSE [uni.name, studyAt.classYear, uniCity.name]
 END  AS
WHERE f.id == friend_VIEW1.f.id
```

```
//final query
SELECT  p.id AS friendId,f.lastName AS friendLastName, distance AS distanceFromPerson,
        p.birthday AS friendBirthday,p.creationDate AS friendCreationDate,
        p.gender AS friendGender,p.browserUsed AS friendBrowserUsed,
        p.locationIp AS friendLocationIp,p.email AS friendEmails,
        p.speaks AS friendSpeaks,friendCityName AS friendCityName,companies AS friendCompanies,
        universities as friendUniversities
OPTIONAL MATCH (f:friend)-[workAt:WORK_AT]->(company:Company)-[:IS_LOCATED_IN]->(companyCountry:Country)
        CASE company.name
        WHEN null THEN null
        ELSE [company.name, workAt.workFrom, companyCountry.name]
        END AS companies
        WHERE f.id = friend_VIEW1.id
```

70: Fianl Query

```

    ELSE   NULL

    RETURN
        f.id AS personId,
        f.firstName AS personFirstName,
        f.lastName AS personLastName,
        m.id AS postID,
        result AS postOrCommentContent,
        m.creationDate AS postDate
    ORDER BY
        postOrCommentContent DESC,
        toInteger(postOrCommentId) ASC
    LIMIT 20
```

Query 2 is a simple query in which there is a path traversal from person to person between whom an interaction happened. From the node Person, a traversal is traversed to another node person along with the label of edge KNOWS. From there on a left path traversal is carried out from the node message to the friend along the edge path `HAS_CREATOR`. The query returns the id, name of the people between who an interaction is carried out, and the date of the post as well as its ID.

7.2.3 Interactive Complex 3

At first a view is created to select the nodes countryXname and countryYName and the person

```
CREATE QUERY city_VIEW1 AS (
FROM LDBC_SNB
```

```
SELECT f.id AS personId,f.firstName AS personFirstName,f.lastName AS personLastName,
        m.id AS postID,result AS postOrCommentContent,m.creationDate AS postDate
FROM LDBC_SNB
MATCH (P:Person {id: $personId })-[:KNOWS]-(f:Person)<-[:HAS_CREATOR]-(m:Message)
      WHERE message.creationDate <= $maxDate
      CASE :
            m.content <> ""
            THEN
                    result = coalesce(m.content,m.imageFile)

            ELSE   NULL


            ORDER BY
                postOrCommentContent DESC,
                toInteger(postOrCommentId) ASC
            LIMIT 20
```

71: ToDo

```
MATCH (x:Country {name: $countryXName }),
      (y:Country {name: $countryYName }),
      (p:Person {id: $personId })
RETURN person
       ,x AS countryA
       ,y AS countryB

LIMIT 1
)
```

In `city_VIEW1` the path between city and country nodes is traversed right along the path `IS_PART_OF`. From the previous view we do need the variables countryA and countryB, to see in which cities a person is or has been After the path traversal the cities a person has been to are returned with their respective country.

```
CREATE QUERY city_VIEW2 AS (
FROM country_VIEW1
MATCH (c:City)-[:IS_PART_OF]->(c:Country)
WHERE country IN [countryA, countryB]
RETURN person
       ,countryA
       ,countryB
       ,COLLECT(c) AS cities
)
```

In the view hereunder, we have a path traversal from the predicate p indicating the node person to a city along the edge with the label `IS_LOCATED_IN`. After the path traversal, the friend its id is returned along with the country.

```
CREATE QUERY city_VIEW3 AS (
FROM city_VIEW2
MATCH (p:Person where  p.id <> f:friend.id)-[:IS_LOCATED_IN]
     ->(c:City WHERE c.id <> c.id)
WHERE p.id == country_VIEW1.p.id
RETURN DISTINCT f
                ,countryA
                ,countryB
)
```

In the query below there are two distinct path traversals. In the first path traversal, there is a left path traversal from the node message to a friend to see to which individual the message belongs, and traverses along the edge with the label HAS_CREATOR. The second path traversal is a right path traversal to retrieve the country from where the message is sent .After traversal, the id is retrieved from the friend and to the related country a value of 1 is assigned if the message is sent from the same country as the location of a person or 0 if the message is sent from a different location.

```
FROM LDBC_SNB,city_VIEW3,city_VIEW2,city_VIEW1
MATCH (f:Friend)<-[:HAS_CREATOR]-(m:message),
      (m:message)-[:IS_LOCATED_IN]->(c:country)
WHERE $endDate > m.creationDate >= $startDate AND
      country IN [countryA, countryB] AND city_VIEW3.f.id == f.id
RETURN f,
     CASE WHEN country=countryA THEN 1 ELSE 0 END AS messageA,
     CASE WHEN country=countryB THEN 1 ELSE 0 END AS messageB
     sum(messageA) AS countA, sum(messageB) AS countB

GROUP BY f.id
HAVING countA >0

AND
       countB>0
```

The last query is a union of union of all the views together that retrieves the respective variables

```
% CALL {
% city_VIEW1
% UNION
% city_VIEW2
% UNION
% city_VIEW3
% RETURN country_VIEW1.f.id AS friend,
%       country_VIEW2.f.firstName AS friend_FirstName,
%       country_VIEW2.f.lastName AS friend_LastName,
%       countA,
%       countB,
%       countA + countA AS ABCount
% }
% ORDER BY ABCount DESC
%          , friend ASC
% LIMIT 20


FROM LDBC_SNB,city_VIEW3,city_VIEW2,city_VIEW1
MATCH (f:Friend)<-[:HAS_CREATOR]-(m:message),
      (m:message)-[:IS_LOCATED_IN]->(c:country)
WHERE $endDate > m.creationDate >= $startDate AND
      country IN [countryA, countryB] AND city_VIEW3.f.id == f.id
// RETURN f,
     CASE WHEN country=countryA THEN 1 ELSE 0 END AS messageA,
     CASE WHEN country=countryB THEN 1 ELSE 0 END AS messageB
//      sum(messageA) AS countA, sum(messageB) AS countB

// GROUP BY f.id
HAVING countA >0 AND countB>0

// CALL {
// city_VIEW1
// UNION
// city_VIEW2
// UNION
// city_VIEW3

// Perhahps we can remove the call Option

RETURN country_VIEW1.f.id AS friend,
       country_VIEW2.f.firstName AS friend_FirstName,
       country_VIEW2.f.lastName AS friend_LastName,
       sum(messageA) AS countA
       ,sum(messageB) AS countB
       countA + countA AS ABCount
// }
ORDER BY ABCount DESC
         , friend ASC
LIMIT 20
```

```
```

```
Select  person,countryX,countryY
FROM LDBC_SNB
MATCH (countryX:Country {name: $countryXName }),
      (countryY:Country {name: $countryYName }),
      (person:Person {id: $personId })
LIMIT 1
```

```
SELECT  person,countryX,countryY,COLLECT(city) AS cities
FROM LDBC,country_VIEW
MATCH (city:City)-[:IS_PART_OF]->(country:Country)
WHERE country IN [countryX, countryY]
```

```
SELECT  DISTINCT person2,countryX,countryY
FROM  LDBC_SNB,part_of_VIEW
MATCH (person)-[:KNOWS]{1,2}-(person2:person)-[:IS_LOCATED_IN]->(city)
WHERE person=person2 AND NOT city IN citiy
```

```
SELECT person2.id AS person2Id, person2.firstName AS person2FirstName
      ,person2.lastName AS person2LastName,sum(messageX) AS xCount
      ,sum(messageY) AS yCount,xCount + yCount AS xyCount
FROM LDBC_SNB,city1_VIEW,city2_VIEW,city3_VIEW
MATCH (person2)<-[:HAS_CREATOR]-(message),
      (message)-[:IS_LOCATED_IN]->(country)
WHERE $endDate > message.creationDate >= $startDate AND
      country IN [countryX, countryY]
CASE WHEN country=countryX
     THEN 1
     ELSE 0
     END AS messageX,
CASE WHEN country=countryY
     THEN 1
     ELSE 0
     END AS messageY
WHERE xCount>0 AND yCount>0
ORDER BY xyCount DESC, person2Id ASC
LIMIT 20
```

75: Final Query

7.2.4 Interactive Complex 4

```
FROM LDBC_SNB
MATCH (p:Person {id: $personId})
      -[:KNOWS]-(f:Person) <-[:HAS_CREATOR]-
       (post:Post)-[:HAS_TAG]->(tag)
CASE $tag :
       WHEN $endDate > post.creationDate >= $startDate THEN 1
       ELSE 0
     END AS valid,
     CASE
       WHEN $startDate > post.creationDate THEN 1
       ELSE 0
     END AS inValid
WHERE countOfPost>0
AND inValidCountOfPost=0
RETURN tag.name AS tagName,
       ,sum(valid) AS countOfPost
       ,sum(inValid) AS inValidCountOfPost
ORDER BY postCount DESC
         ,tagName ASC
LIMIT 10
```

The construction of a view is omitted since there is only one MATCH clause. In the MATCH clause a path is traversed from the node Person to another person traversing the first edge label named KNOWS. Afterward, a left traversal is done from the node post to person along the label named `HAS_CREATOR`. From the node post, there is another traversal going out or coming in from post to tag and is a right traversal along the edge to tag. In short, this query verifies whether a new post or comment is interchanged between two people.

```
SELECT tag.name AS tagName,sum(valid) AS countOfPost,sum(inValid) AS inValidCountOfPost
FROM LDBC_SNB
    MATCH (p:Person {id: $personId })-[:KNOWS]-(f:Person) <-[:HAS_CREATOR]-(post:Post)-[:HAS_TAG]->(tag)
       CASE $tag :
       WHEN $endDate > post.creationDate >= $startDate THEN 1
       ELSE 0
     END AS valid,
     CASE
       WHEN $startDate > post.creationDate THEN 1
       ELSE 0
     END AS inValid
WHERE countOfPost>0 AND inValidCountOfPost=0
ORDER BY postCount DESC
         ,tagName ASC
LIMIT 10
```

76: TODO

7.2.5 Interactive Complex 5

In the view `know_VIEW`, the path of KNOWS is traversed between person and friend of lengths of between 1 and 2 and returns the distinct friends.

```
CREATE QUERY know_VIEW as (
MATCH (p:Person { id: $personId })-[:KNOWS]-(f: friend) {1,2}
WHERE
     person <> f
Return DISTINCT f
)
```

From the view `know_VIEW` the path with the label `HAS_MEMBER` is traversed from the node friend to the node forum. The traversal is the right one. Upon that path, all the friends who have joined a forum are accumulated and returned.

```
CREATE QUERY forum_VIEW AS (
FROM know_VIEW
MATCH (f)<-[m:HAS_MEMBER]-(forum)
WHERE
    m.joinDate > $minDate

AND know_VIEW.f.id == f.id
RETURN forum
      ,COLLECT(f) AS friends
)
```

In the Query below from the edge forum, there is a path traversal along the edge of label `CONTAINER_OF` to post (left traversal). Another left traversal is done from the node post to a friend along with the edge label `HAS_CREATOR`, to see whether a person posted in the forum. Eventually, the name of the forum and the number of posts posted by a person is returned.

```
FROM know_VIEW,forum_VIEW
OPTIONAL MATCH (f)<-[:HAS_CREATOR]-(post)<-[:CONTAINER_OF]-(forum)
WHERE
    f IN friends
RETURN
    f.id as person
    forum.title AS nameForum,
    count(post) AS countPost
ORDER BY
    postCount DESC,
    forum.id ASC
LIMIT 20
```

```
SELECT  DISTINCT f
MATCH (p:Person { id: $personId })-[:KNOWS]-(f: friend) {1,2}
WHERE
     person <> f
```

```
SELECT forum, COLLECT(f) AS friends
FROM know_VIEW
MATCH (f)<-[m:HAS_MEMBER]-(forum)
WHERE
    m.joinDate > $minDate
AND know_VIEW.f.id == f.id
```

7.2.6 Interactive Complex 6

In the view below `tag_VIEW`, all the tags-id are retrieved and returned. Within the MATCH clause, there is no path traversal just selection of the specific node Tag

```
Create Query tag_VIEW AS (
MATCH (knowTag:Tag { name: $tagName })
WITH knowTag.id as knownTagId
)
```

The query `knowTag_VIEW` traverses from the node person to friend and does at most 2 undirected traversals between 1 and 2. The traversal along the path with edge label: KNOWS makes sure that each person is not a friend within himself or herself. In the end, the relatedTagID and the friendID information are returned.

```
//final Query
SELECT f.id as person, forum.title AS nameForum, count(post) AS countPost
FROM know_VIEW,forum_VIEW
OPTIONAL MATCH (f)<-[:HAS_CREATOR]-(post)<-[:CONTAINER_OF]-(forum)
WHERE
    f IN friends
ORDER BY
    postCount DESC,
    forum.id ASC
LIMIT 20
```

79: Final Query

```
CREATE Query knowTag_VIEW as (
FROM tag_VIEW
MATCH (p:Person { id: $personId })-[:KNOWS]-(f: friend){1,2}
WHERE  p<>f AND tag_VIEW.knownTagId == knownTagId
RETURN
    knownTagId,
    (distinct f) as friends
)
```

In the final query, the path traversal is executed from the friend node to the post node along the path `HAS_CREATOR`. There is also path traversal from post to TagID. Do note that there are two path traversals from post to Tag, indicating that they are two different tags. At last, the query returns the tag names and the number of posts posted by each person.

```
FROM tag_VIEW, knowTag_VIEW
MATCH (f)<-[:HAS_CREATOR]-(post:Post),
        (post)-[:HAS_TAG]->(t1:Tag{id: knownTagId}),
        (post)-[:HAS_TAG]->(t2:Tag)
    WHERE  t1 <> t2
RETURN
   t.name as nametag,
 count(post) as countPost
ORDER BY
    nametag DESC,
    countPost ASC
LIMIT 10
```

```
SELECT  knowTag.id as knownTagId
FROM LDBC_SNB
     MATCH (knowTag:Tag { name: $tagName })
```

```
SELECT knownTagId, (distinct f) as friends
FROM tag_VIEW
     MATCH (p:Person { id: $personId })-[:KNOWS]-(f: friend){1,2}
     WHERE  p<>f AND tag_VIEW.knownTagId == knownTagId
```

```
SELECT  t.name as nametag,count(post) as countPost
FROM tag_VIEW, knowTag_VIEW
MATCH (f)<-[:HAS_CREATOR]-(post:Post),
      (post)-[:HAS_TAG]->(t1:Tag{id: knownTagId}),
      (post)-[:HAS_TAG]->(t2:Tag)
WHERE  t1 <> t2
ORDER BY
    nametag DESC,
    countPost ASC
LIMIT 10
```

82: final query

7.2.7 Interactive Complex 7

For the query above, there is no view necessary since it consists of one query. There are two match clauses. In the first match clause, a left traversal is done from comment to messageId, and also a right traversal to the node person along with the edge label `HAS_CREATOR`. In the second match, which is an optional MATCH, a right traversal is carried out from the message node to the person, and then another right traversal to the person node along the path `HAS_CREATOR`. From the node person, there is another traversal to the node person. This seems confusing, but the query verifies whether a reply is given from person A to person B. In addition, a verification is done to see whether two people know each other.

```
FROM LDBC_SNB
MATCH (m:Message {id: $messageId })<-[:REPLY_OF]-(c:Comment)-
      [:HAS_CREATOR]->(p:Person)
OPTIONAL MATCH (m)-[:HAS_CREATOR]->(a:Person)-[r:KNOWS]-(p)
RETURN c.id AS commentId,
        c.content AS Content,
        c.creationDate AS CreationDate,
        p.id AS AuthorId,
        p.firstName AS replyFirstName,
        p.lastName AS replyLastName,
        CASE r
            WHEN null THEN false
            ELSE true
       END AS KnowsOriginalMessageAuthor
ORDER BY creationDate DESC
         ,AuthorId
```

7.3 Interactive 7

```
SELECT c.id AS commentId, c.content AS Content, c.creationDate AS CreationDate, p.id AS AuthorId,
        p.firstName AS replyFirstName,p.lastName AS replyLastName,
        CASE r
            WHEN null THEN false
            ELSE true
       END AS KnowsOriginalMessageAuthor
FROM LDBC_SNB
MATCH (m:Message {id: $messageId })<-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p:Person)
OPTIONAL MATCH (m)-[:HAS_CREATOR]->(a:Person)-[r:KNOWS]-(p)
ORDER BY creationDate DESC
         ,AuthorId
```

83: final query

7.3.1 Interactive Complex 8

As in Query 7, there construction of a view is omitted as well. Here the match clause has four nodes. First, from p1 indicating the person node, there is a path traversal to the message node, which is a left traversal. Also from the message node, there is a left traversal from comment to message and at last from comment node to person node there is a right traversal along with the `HAS_CREATOR` label. In short, the match clause in principle is a follow-up of the query executed before. The clause looks up at all the replies between two people.

```
FROM LDBC_SNB
MATCH (p1:Person {id: $personId})<-[:HAS_CREATOR]-(:Message)<-[:REPLY_OF]
     -(c:Comment)-[:HAS_CREATOR]->(p2:Person)

RETURN
    p1.id AS personId,
    p1.firstName AS personFirstName,
    p1.lastName AS personLastName,
    c.creationDate AS commentCreationDate,
    c.id AS commentId,
    c.content AS content
ORDER BY
    commentCreationDate DESC,
    commentId ASC
LIMIT 20
```

```
//
SELECT p1.id AS personId, p1.firstName AS personFirstName, p1.lastName AS personLastName,
       c.creationDate AS commentCreationDate,c.id AS commentId,c.content AS content
FROM LDBC_SNB
MATCH (p1:Person {id: $personId})<-[:HAS_CREATOR]-(:Message)<-[:REPLY_OF]
      -(c:Comment)-[:HAS_CREATOR]->(p2:Person)
ORDER BY
    commentCreationDate DESC,
    commentId ASC
LIMIT 20
```

84: ToDo

7.3.2 Interactive Complex 9

The match clause of `friends_VIEW` verifies the friends of friends and makes sure that the friendship is not a reflexive relationship. The path traversal is done to a path of at most 2 lengths, with a minimum of 1 length.

```
CREATE Query friends_VIEW as (
MATCH (p1:Person {id: $personId })-[:KNOWS]-(p2:Person){1,2}
WHERE f<>p1
RETURN  COLLECT (distinct f) as friends
)
```

The query below is a continuation of the one above, and does a right path traversal from the message node to the friend node. The combination of this query, and the constructed view above verifies whether a new message is sent between two friends and the type of message. The RETURN clause returns the details of the owner of that message, and the content of the message

```
FROM friends_VIEW,LDBC_SNB
MATCH (f) <-[:HAS_CREATOR]-(m:Message)
    WHERE m.creationDate < $maxDate
RETURN
    f.id AS personId,
    f.firstName AS FirstName,
    f.lastName AS LastName,
    m.id AS commentId,
    coalesce(m.content,m.imageFile) AS Content,
    m.creationDate AS contentCreationDate
ORDER BY
    contentCreationDate DESC,
    message.id ASC
LIMIT 20
```

```
SELECT  COLLECT (distinct f) as friends
FROM LDBC_SNB
MATCH (p1:Person {id: $personId })-[:KNOWS]-(p2:Person){1,2}
WHERE f<>p1
```

```
SELECT  f.id AS personId,f.firstName AS FirstName,f.lastName AS LastName, m.id AS commentId,
        coalesce(m.content,m.imageFile) AS Content,
        m.creationDate AS contentCreationDate
FROM friends_VIEW
    MATCH (f) <-[:HAS_CREATOR]-(m:Message)
    WHERE m.creationDate < $maxDate
ORDER BY
    contentCreationDate DESC,
    message.id ASC
LIMIT 20
```

86: Final Query

7.3.3 Interactive Complex 10

In the first query view below we do two-match clauses. In the first match clause, there is an undirected path traversal from both the person node to the friend node traversed along the path KNOWS that traverses between 1 or 2 paths. In the second MATCH clause, a traversal is done from the person node to the friend node along the path of KNOWS.

```
Create Query friend_VIEW as {
MATCH (p:Person {id: \$personId})-[:KNOWS]-(friend){2,..}
       (friend)-[:IS_LOCATED_IN]->(city:City)
OPTIONAL MATCH   (friend where friend <> person)-[:KNOWS]-(person)
WHERE  (birthday.month=$month AND birthday.day>=21) OR
        (birthday.month=($month%12)+1 AND birthday.day<22)
RETURN
      person
      ,city
      ,friend
      ,datetime({f.birthday}) as birthday
}
```

Do notice that the postExtend_VIEW query is an extension of the post_VIEW. The post_VIEW first returns the friend, city, and posts values through the optional match, which is a right traversal from the post node to the friend node along the path of the edge label HAS_CREATOR.Afterward, the extended view carries out an inner match by traversing the path from person node to another person node. This, initially might not make sense. But this inner traversal is done to see whether person1 might know person2 based on their location, posts, and comments.

```
Create Query post_VIEW as {
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
RETURN friend
       ,city
       ,COLLECT(post) AS posts
       ,person
}
```

```
Create Query postExtend_VIEW as {
FROM post_VIEW
Return city,
     size(posts) AS postCount,
     friend.id AS personId,
     friend.firstName AS personFirstName,
     friend.lastName AS personLastName,
     commonPostCount - (postCount - commonPostCount) AS commonInterestScore,
     friend.gender AS personGender,
     city.name AS personCityName
WHERE EXIST{
  FROM LBC_SNB
  MATCH
  (p)-[:HAS_TAG]->()<-[:HAS_INTEREST]-(person))

  Return  AS commonPostCount
   }
ORDER BY commonInterestScore DESC
        , personId ASC
LIMIT 10
}
```

```
SELECT person,city,friend,datetime({f.birthday}) as birthday
FROM LDBC_SNB
    MATCH (p:Person {id: $personId})-[:KNOWS]-(friend){2,..}
       (friend)-[:IS_LOCATED_IN]->(city:City)
    OPTIONAL MATCH   (friend where friend <> person)-[:KNOWS]-(person)
    WHERE  (birthday.month=$month AND birthday.day>=21) OR
           (birthday.month=($month%12)+1 AND birthday.day<22)
```

```
SELECT friend,city,COLLECT(post) AS posts,person
FROM LDBC_SNB
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
```

7.3.4 Interactive Complex 11

The view retrieves people who are friends. The construction of this is done by first traversing the path with edge label KNOWS from predicate p1 (person node 1) to p2(another person).

The query above traverses the path from company to country name along the path `IS_LOCATED_IN` to the node country,which is a left traversal. There is also an undirected traversal from the friend node towards the company node.In short, the query retrieves the job referrals.

```
SELECT city,size(posts) AS postCount,friend.id AS personId,
       friend.firstName AS personFirstName,friend.lastName AS personLastName,
       commonPostCount - (postCount - commonPostCount) AS commonInterestScore,
       friend.gender AS personGender,city.name AS personCityName
FROM
WHERE EXIST{
  SELECT AS commonPostCount
  FROM LBC_SNB
  MATCH
  (p)-[:HAS_TAG]->()<-[:HAS_INTEREST]-(person)

   }
ORDER BY commonInterestScore DESC
        , personId ASC
LIMIT 10
```

```
Create Query personFriend_VIEW AS (
FROM LDBC_SNB
MATCH (p1:Person {id: \$personId })-[:KNOWS]-(p2:Person){1,2}
WHERE not(person <> p2)
RETURN DISTINCT friend
)
```

```
FROM LDBC_SNB,personFriend_VIEW
MATCH (:Country {name: \$countryName })
      <-[:IS_LOCATED_IN] <- (friend)
      -[workAt:WORK_AT] -(company:Company)
WHERE workAt.workFrom < \$workFromYear
RETURN
        f.id AS personId,
        f.firstName AS personFirstName,
        f.lastName AS personLastName,
        company.name AS organizationName,
        workAt.workFrom AS organizationWorkFromYear
ORDER BY
        organizationWorkFromYear ASC,
        toInteger(personId) ASC,
        organizationName DESC
LIMIT 10
```

```
SELECT  DISTINCT friend
FROM LDBC_SNB
MATCH (p1:Person {id: $personId })-[:KNOWS]-(p2:Person){1,2}
WHERE (person <> p2)
```

7.3.5 Interactive Complex 12

In the query above we have the MATCH clause that does a traversal from the baseTagclass that comes from the tag node in a left manner and traveled through the `HAS_TYPE` label. During traversal, at least one path is traveled. In the second query, we have the traversal from the comment node towards the friend node first through the label `HAS_CREATOR`. From the friend nodes, there is also an undirected traversal towards the person node. Moreover, there are two right traversals from the comment node towards the post node, which is the first through the POST label.

```
SELECT f.id AS personId,f.firstName AS personFirstName,f.lastName AS personLastName,company.name AS organizationName,workAt.workF
FROM LDBC_SNB,personFriend_VIEW
MATCH (:Country {name: $countryName }) <-[:IS_LOCATED_IN] <- (friend) -[workAt:WORK_AT] -(company:Company)
WHERE workAt.workFrom < $workFromYear
ORDER BY
        organizationWorkFromYear ASC,
        toInteger(personId) ASC,
        organizationName DESC
LIMIT 10
```

91: Final Query

```
FROM LDBC_SNB
MATCH (baseTagClass:TagClass)<-[:HAS_TYPE|IS_SUBCLASS_OF]{0,..}-(tag:Tag)
WHERE tag.name = $tagClassName OR baseTagClass.name = $tagClassName
RETURN collect(tag.id) as tags
```

```
MATCH (:Person {id: \$personId})-[:KNOWS]-(friend:Person)
     <-[:HAS_CREATOR]-(comment:Comment)
     -[:REPLY_OF]->(:Post)-[:HAS_TAG]->(tag:Tag)
WHERE tag.id in tags
RETURN
    friend.id AS personId,
    friend.firstName AS personFirstName,
    friend.lastName AS personLastName,
    COLLECT(DISTINCT tag.name) AS tagNames,
    COUNT(DISTINCT comment) AS replyCount
ORDER BY
    replyCount DESC,
    toInteger(personId) ASC
LIMIT 20
```

The second traversal is the traversal from the comment node towards the post node, that travels through the path `REPLY_OF`. Afterward, a second traversal is done from the post node towards the tag node through the `HAS_TAG` path.

```
SELECT collect(tag.id) as tags
FROM LDBC_SNB
      MATCH (baseTagClass:TagClass)<-[:HAS_TYPE|IS_SUBCLASS_OF]{0,..}-(tag:Tag)
WHERE tag.name = $tagClassName OR baseTagClass.name = $tagClassName

SELECT friend.id AS personId,friend.firstName AS personFirstName,friend.lastName AS personLastName, COLLECT(DISTINCT tag.name) AS
FROM
     MATCH (:Person {id: $personId })-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-(comment:Comment)-[:REPLY_OF]->(:Post)-[:HAS_TAG]-
WHERE tag.id in tags
ORDER BY
    replyCount DESC,
    toInteger(personId) ASC
LIMIT 20
```

92: ToDo

7.3.6 Interactive Complex 13

```
FROM LDBC_SNB
MATCH(p1:Person {id: $person1Id}),(p2:Person {id: $person2Id}),
    path = shortest((p1)-[:KNOWS]-(p2)){*}
RETURN
    CASE path IS NULL
        WHEN true THEN -1
        ELSE length(path)
    END AS shortestPathLength
```

The shortest path available **SCHRIJF DIE PAGINA NUMMER OP OF REFERENCE** functionality is used here and is traversed at most between 0 or more repetitions. As a final result, the length of the path is returned.

```
SELECT  CASE path IS NULL  WHEN true THEN -1 ELSE length(path)
        END AS shortestPathLength
FROM LDBC_SNB
     MATCH( p1:Person {id: $person1Id}),(p2:Person {id: $person2Id}),
     path = SHORTEST((p1)-[:KNOWS]-(p2)){*}
```

93: ToDo

7.3.7 Interactive Short 1

The query below returns the results of a single person by traversing the path labeled IS_LOCATED_INto the left from person to city node.

```
MATCH (p1:Person {id: $personId })-[:IS_LOCATED_IN]->(c:City)
RETURN
    p1.firstName AS firstName,
    p1.lastName AS lastName,
    p1.birthday AS birthday,
    p1.locationIP AS locationIP,
    p1.browserUsed AS browserUsed,
    p1.id AS cityId,
    p1.gender AS gender,
    p1.creationDate AS creationDate
```

// IS2. Recent messages of a person

```
SELECT p1.firstName AS firstName,p1.lastName AS lastName,p1.birthday AS birthday,p1.locationIP AS locationIP,p1.browserUsed AS br
FROM LDBC_SNB
MATCH (p1:Person {id: $personId })-[:IS_LOCATED_IN]->(c:City)
```

94: Profile of a person

7.3.8 Interactive Short 2

In the query below the view carries out a left traversal from the unlabeled message node to the person node in a left manner. This traversal is done to obtain all the people who are the creator of a message.

```
Create Query message_VIEW as (
MATCH (:Person {id: $personId})<-[:HAS_CREATOR]-(:message)
Return
 message,
 message.id AS messageId,
 message.creationDate AS messageCreationDate
ORDER BY messageCreationDate DESC, messageId ASC
LIMIT 10
)
```

```
From message_VIEW, LDBC_SNB
MATCH (message)-[:REPLY_OF]->(post:Post)-[:HAS_CREATOR]->(person)
RETURN  messageId,
 ,coalesce(message.imageFile,message.content) AS messageContent
,messageCreationDate
 ,post.id AS postId,
 ,person.id AS personId,
 ,person.firstName AS FirstName,
 ,person.lastName AS LastName
ORDER BY messageCreationDate DESC
        ,messageId ASC
```

From the Match clause above in query 2, there is a traversal from the message node to the post node traveled along with the `REPLY_OF` label. Do note that this is a left path traversal. After the left path traversal, there is a left traversal from the person node to the post node along the path of `HAS_CREATOR` to the person node. This traversal looks for the people that replied to a message. // IS2. Recent messages of a person

```
//  Query message_VIEW
SELECT message,message.id AS messageId,message.creationDate AS messageCreationDate
MATCH (:Person {id: $personId})<-[:HAS_CREATOR]-(:message)
ORDER BY messageCreationDate DESC, messageId ASC
LIMIT 10


\begin{listing}[!ht]
\begin{minted}
[
frame=lines,
framesep=2mm,
baselinestretch=1.2,
bgcolor=LightGray,
fontsize=\footnotesize,
linenos
]
{cypher}
SELECT  messageId,coalesce(message.imageFile,message.content) AS messageContent ,messageCreationDate,post.id AS postId,person.id
From LDBC_SNB,...
MATCH (message)-[:REPLY_OF]->(post:Post)-[:HAS_CREATOR]->(person)
ORDER BY messageCreationDate DESC
        ,messageId ASC
```

95: ToDo

7.3.9 Interactive Short 3

This query retrieves all the friends of a person through the undirected traversal between the node person and the another node friend by traversing the path KNOWS.

7.3.10 Interactive Short 4

The query selects the message node and afterwards returns the creation date, that a message was created and message content. The message content consist out of a COALESCENCE clause. This mean that wehn content is empty then the imageFile is chosen as the messageContent,and vice versa. **TODO.tex**

7.3.11 Interactive Short 5

```
FROM LDBC_SNB
MATCH (n:Person {id: \$personId })-[r:KNOWS]-(f:friend)
RETURN
    f.id AS personId,
    f.firstName AS firstName,
    f.lastName AS lastName,
    r.creationDate AS friendshipDate
ORDER BY
    friendshipDate DESC,
    CAST(personId AS INTEGER) ASC
```

```
SELECT f.id AS personId,f.firstName AS firstName,f.lastName AS lastName,r.creationDate AS friendshipDate
FROM LDBC_SNB
     MATCH (n:Person {id: $personId })-[r:KNOWS]-(f:friend)
ORDER BY
    friendshipDate DESC,
    CAST(personId AS INTEGER) ASC
```

96: Friends of a person

```
FROM LDBC_SNB
MATCH (message:Message {id:  \$messageId })
RETURN
    message.creationDate as messageCreationDate,
    COALESCENCE (m.content, m.imageFile) as messageContent
```

97: message contents

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

98: ToDo

```
FROM LDBC_SNBI
MATCH (m:Message {id: \$messageId })-[:HAS_CREATOR]->(p1:Person)
RETURN
    p1.id AS personId,
    p1.firstName AS firstName,
    p1.lastName AS lastName
```

The query traverses from the node message to the person in a left manner along the path `HAS_CREATOR` in retrieving the details of the creator of a message.

```
SELECT
FROM LDBC_SNB
    MATCH (forum:Forum {id: $forumId}), (person:Person {id: $personId})
    CREATE (forum)-[:HAS_MEMBER {joinDate: $joinDate}]->(person)
```

99: ToDo

7.3.12 Interactive Short 6

```
FROM LDBC_SNBI
OPTIONAL MATCH (m:Message {id: $messageId})-[:REPLY_OF]{0,}->(p:Post)<-[:CONTAINER_OF]-(f:Forum)-[:HAS_MODERATOR]->(p1:Person)
RETURN
    f.id AS forumId,
    f.title AS forumTitle,
    mod.id AS moderatorId,
    mod.firstName AS FirstName,
    mod.lastName AS LastName
```

An optional match is used in the query rather than an if clause in GSQL.The use of an OPTIONAL MATCH will result in a returned NULL value if there is no value found for such. From the message node, there is a right traversal to the post node along the path of `REPLY_OF`. In addition a left path traversal is present, and travels along the path `CONTAINER_OF` from the forum node to the post node. Additionally another right traversal is done from the forum node towards the person node, through the path `HAS_MODERATOR` . Since Comments are not directly contained in Forums, for Comments, the Forum containing the original Post in the thread to which the Comment is replying is returned. // IS6. Forum of a message

```
SELECT f.id AS forumId, f.title AS forumTitle, mod.id AS moderatorId, mod.firstName AS FirstName, mod.lastName AS LastName
FROM LDBC_SNBI
     OPTIONAL MATCH (m:Message {id: $messageId })-[:REPLY_OF]{0,}->(p:Post)<-[:CONTAINER_OF]-(f:Forum)-[:HAS_MODERATOR]->(p1:Pers
```

100: ToDo

7.3.13 Interactive Short 7

In the match clause, the query looks at whether there is a reply to a message or not by carrying out a left path traversal from the comment node to the message node along the path of `REPLY_OF`. From the comment node also a right traversal is done to the person node through the path of `HAS_CREATOR`. Furthermore, there is an OPTIONAL MATCH from another person to another one to see if the message exchanged between two people do know each other.

// IS7. Replies of a message

7.3.14 Interactive Insert 1

In the query below a selection is done on the city node. After selecting that node a temporary view is constructed, in which the person's details are included. Also in the view a path is traversed from the person node towards the city node. The traversal is executed through the path named `IS_LOCATED_IN` . After retrieval of the id's with their respective tags, the following query selects the results obtained, which are id's of the tag and creates another node view which is traversed from the person to the tag node, along the path named `HAS_INTEREST`

In the view `interest_VIEW` there is a selection of the organization node. Also in that sub-query, another node view is created from the node p where there is a traversal from the person node to the organization node, at which a traversal is done along the path of `STUDY_AT`. In addition, there is another path created from the person node to the organization node, and a path traversal is done through the `WORKS_AT` path.

```
FROM LDBC_SNBI
MATCH (m:Message {id: \$messageId})<-[:REPLY_OF]-(c:Comment)
      -[:HAS_CREATOR]->(p:Person)
OPTIONAL MATCH (m)-[:HAS_CREATOR]->(p2:Person)-[r:KNOWS]-(p1)
RETURN c.id AS commentId,
c.content AS content,
c.creationDate AS creationDate,
p.id AS AuthorId,
p.firstName AS AuthorFirstName,
p.lastName AS AuthorLastName,
CASE r
     WHEN null THEN false
    ELSE true
        END AS replyAuthorKnowsOriginalMessageAuthor
ORDER BY commentCreationDate DESC, AuthorId
```

```
SELECT c.id AS commentId,c.content AS content,c.creationDate AS creationDate,p.id AS AuthorId,p.firstName AS AuthorFirstName,p.la
       CASE r WHEN null THEN false ELSE true  END AS replyAuthorKnowsOriginalMessageAuthor
FROM LDBC_SNBI
     MATCH (m:Message {id: $messageId })<-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p:Person)
     OPTIONAL MATCH (m)-[:HAS_CREATOR]->(p2:Person)-[r:KNOWS]-(p1)
ORDER BY commentCreationDate DESC, AuthorId
```

101: ToDo

```
CREATE QUERY tag_VIEW AS {
FROM LDBC_SNBI
MATCH (c:City {id: $cityId})
CREATE (p:Person {
    id: $personId,
    firstName: $personFirstName,
    lastName: $personLastName,
    gender: $gender,
    birthday: $birthday,
    creationDate: $creationDate,
    locationIP: $locationIP,
    browserUsed: $browserUsed,
    languages: $languages,
    email: $emails
  })-[:IS_LOCATED_IN]->(c)
RETURN  $tagIds AS tagId
}
```

Can we construct it like this? the workAt variable (ask)

7.3.15 Interactive Insert 2

TODO

7.3.16 Interactive Insert 3

The query above first selects the person node and the comment node and then creates path traversal from the

```
Create Query as  interest_VIEW AS {
FROM tag_VIEW
  MATCH (t:Tag {id: tagId})
  CREATE (p)-[:HAS_INTEREST]->(t)
 Return p, count(*) AS times,
        \$studyAt AS place
}
```

```
FROM interest_VIEW,LDBC_SNB
MATCH (o:Organisation {id: s[0]})
CREATE (p)-[:STUDY_AT {classYear: s[1]}]->(o)
Return p
      ,count(*) AS times2
      ,$workAt AS w
```

person node towards the comment node in a right traversal manner carried out through the constructed path named LIKES.

7.3.17 Interactive Insert 4

The construction of views is omitted. The first query selects the person node and then creates a path traversal from the forum node towards the person node, along the path of `HAS_MODERATOR`. From the returned values of the query above, query 2 continues or starts with the tag node, and from there on a traversal is created along the path of `HAS_TAG` from the forum node towards the tag node.

```
MATCH (tag:Tag {id: tagId})
CREATE (f)-[:HAS_TAG]->(tag)
```

7.3.18 Interactive Insert 5

A selection is made in the match clause, selecting the forum node and the person node. After that, a path is created from the forum node towards the person node in a right manner along the path of `HAS_MEMBER`.

7.3.19 Interactive Insert 6

In here the query first selects the message node and from there on a right path traversal is executed, through the path `REPLY_OF`. A minimum of zero paths are traversed. Towards the post node there is an incoming path traversal from the forum node, and traverses through the path named `CONTAINER_OF`. This path is a LUR.Another path traversal is present and happens from the forum node towards the person node through the path named `HAS_MODERATOR`

```
MATCH (comp:Organisation {id: w[0]})
  CREATE (p)-[:WORKS_AT {workFrom: w[1]}]->(comp)
```

```
SELECT  $tagIds AS tagId
FROM LDBC_SNBI
     MATCH (c:City {id: $cityId})
     CREATE (p:Person {id: $personId,firstName: $personFirstName,lastName: $personLastName,gender: $gender,birthday: $birthday,cre
         browserUsed: $browserUsed,languages: $languages,email: $emails })-[:IS_LOCATED_IN]->(c)
```

```
SELECT p, count(*) AS times,
FROM
    MATCH (t:Tag {id: tagId})
    CREATE (p)-[:HAS_INTEREST]->(t)
        $studyAt AS place
```

```
SELECT p, count(*) AS times,$studyAt AS place
FROM
    MATCH (t:Tag {id: tagId})
    CREATE (p)-[:HAS_INTEREST]->(t)
```

104: ToDo

```
SELECT p,count(*) AS times2,$workAt AS w MATCH (comp:Organisation {id: w[0]})
FROM interest_VIEW,LDBC_SNB
     MATCH (o:Organisation {id: s[0]})
     CREATE (p)-[:STUDY_AT {classYear: s[1]}]->(o)
     CREATE (p)-[:WORKS_AT {workFrom: w[1]}]->(comp)
```

105: ToDo

```
SELECT
From LDBC_SNB
     MATCH (person:Person {id: $personId}), (comment:Comment {id: $commentId})
     CREATE (person)-[:LIKES {creationDate: \$creationDate}]->(comment)
```

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

7.3.20 Interactive Insert 7

In the MATCH clause above, the country, person, and message node are selected. After selection, a left traversal is done from the comment node towards the author node along the path of `HAS_CREATOR`.From the comment node,

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

106: ToDo

```
FROM LDBC_SNB
MATCH (p:Person {id: $moderatorPersonId})
CREATE (f:Forum {id: $forumId, title: $forumTitle, creationDate: $creationDate})-[:HAS_MODERATOR]->(p)
Return $tagIds AS tagId
```

```
SELECT $tagIds AS tagId
FROM LDBC_SNB
    MATCH (p:Person {id: $moderatorPersonId})
    CREATE (f:Forum {id: $forumId, title: $forumTitle, creationDate: $creationDate})-[:HAS_MODERATOR]->(p)

SELECT ...
FROM ...
MATCH (tag:Tag {id: tagId})
CREATE (f)-[:HAS_TAG]->(tag)
```

107: ToDo

```
FROM LDBC_SNB
MATCH (forum:Forum {id: $forumId}), (person:Person {id: $personId})
CREATE (forum)-[:HAS_MEMBER {joinDate: $joinDate}]->(person)
```

```
SELECT
FROM LDBC_SNB
    MATCH (forum:Forum {id: $forumId}), (person:Person {id: $personId})
CREATE (forum)-[:HAS_MEMBER {joinDate: $joinDate}]->(person)
```

108: ToDo

there is a right traversal towards the message node through the path named `REPLY_OF`. In addition, there is a second MATCH clause that does a traversal from the country node towards the country node along the path `IS_LOCATED_IN`.

In the query above there is no from clause needed since it latches on the results of the previous query. The MATCH clause selects the tag's returned in the query and creates a path from the country node towards the the tag node.

```
FROM  LDBC_SNB
MATCH (m:Message {id: \$messageId })-[:REPLY_OF*]{0,..}->(p:Post)<-[:CONTAINER_OF]-(f:Forum)-[:HAS_MODERATOR]->(mod:Person)
RETURN
    f.id AS forumId,
    f.title AS forumTitle,
    mod.id AS moderatorId,
    mod.firstName AS moderatorFirstName,
    mod.lastName AS moderatorLastName
```

109: Moderator

```
SELECT
MATCH (p:Person {id: $authorPersonId}), (country:Country {id: $countryId}), (forum:Forum {id: $forumId})
CREATE (author)<-[:HAS_CREATOR]-(pm:Post:Message {
    id: $postId,creationDate: $creationDate,locationIP: $locationIP, browserUsed: $browserUsed,content: CASE $content WHEN '' THE
    length: $length})<-[:CONTAINER_OF]-(forum), (p)-[:IS_LOCATED_IN]->(country)
RETURN $tagIds AS tagId
```

110: ToDo

```
SELECT ...
FROM ...
     MATCH (t:Tag {id: tagId})
CREATE (p)-[:HAS_TAG]->(t)
```

111: ToDo

```
FROM LDBC_SNB
MATCH
  (p1:Person {id:\$authorPersonId}),
  (c:Country {id: \$countryId}),
  (m:Message {id: \$replyToPostId + \$replyToCommentId + 1}) // \$replyToCommentId is -1 if the message is a reply to a post and

CREATE (author)<-[:HAS_CREATOR]-(c:Comment:Message {
    id: \$commentId,
    creationDate: \$creationDate,
    locationIP: $locationIP,
    browserUsed: $browserUsed,
    content: \$content,
    length: \$length
  })-[:REPLY_OF]->(message),
  (c)-[:IS_LOCATED_IN]->(country)
Return comment
      ,\$tagIds AS tagId
```

112:

```
  FROM ...
  MATCH (t:Tag {id: tagId})
  CREATE (c)-[:HAS_TAG]->(t)
```

```
SELECT c,$tagIds AS tagId
FROM LDBC_SNB
     MATCH (author:Person {id: $authorPersonId}),(country:Country {id: $countryId}),
           (message:Message {id: $replyToPostId + $replyToCommentId + 1})
       //$replyToCommentId is -1 if the message is a reply to a post and vica versa (see spec)
     CREATE (author)<-[:HAS_CREATOR]-(c:Comment:Message {id: $commentId,creationDate: $creationDate,locationIP: $locationIP,brows
            ,content: $content,length: $length})
            -[:REPLY_OF]->(message),
      (c)-[:IS_LOCATED_IN]->(country)
```

```
SELECT ...
FROM .... //grab the tagID from the results retreived from above
    MATCH (t:Tag {id: tagId})
    CREATE (c)-[:HAS_TAG]->(t)
```

113: ToDo

7.3.21 Interactive Insert 8

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

In the match clause the person node is selected,and a relationship is created between the selected nodes and the label :KNOWS is given.

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

114: ToDo