G-CORE
A Core for Future Graph Query Languages

Designed by the LDBC Graph Query Language Task Force*

RENZO ANGLES, Universidad de Talca
MARCELO ARENAS, PUC Chile
PABLO BARCELÓ, DCC, Universidad de Chile
PETER BONCZ, CWI, Amsterdam
GEORGE H. L. FLETCHER, Technische Universiteit Eindhoven
CLAUDIO GUTIERREZ, DCC, Universidad de Chile
TOBIAS LINDAAKER, Neo4j
MARCUS PARADIES, SAP SE
STEFAN PLANTIKOW, Neo4j
JUAN SEQUEDA, Capsenta
OSKAR VAN REST, Oracle
HANNES VOIGT, Technische Universität Dresden

We report on a community effort between industry and academia to shape the future of graph query languages. We argue that existing graph database management systems should consider supporting a query language with two key characteristics. First, it should be composable, meaning, that graphs are the input and the output of queries. Second, the graph query language should treat paths as first-class citizens. Our result is G-CORE, a powerful graph query language design that fulfills these goals, and strikes a careful balance between path query expressivity and evaluation complexity.

PREAMBLE

G-CORE is a design by the LDBC Graph Query Language Task Force, consisting of members from industry and academia, intending to bring the best of both worlds to graph practitioners.

*This paper is the culmination of 2.5 years of intensive discussion between the LDBC Graph Query Language Task Force and members of industry and academia. We thank the following organizations who participated in this effort: Capsenta, HP, Huawei, IBM, Neo4j, Oracle, SAP and Sparsity. We also thank the following people for their participation: Alex Averbuch, Hassan Chafi, Irini Fundulaki, Alastair Green, Josep Lluis Larriba Pey, Jan Michels, Raquel Pau, Arnau Prat, Tomer Sagi and Yinglong Xia.

Authors' addresses: Renzo Angles, Universidad de Talca; Marcelo Arenas, PUC Chile; Pablo Barceló, DCC, Universidad de Chile; Peter Boncz, CWI, Amsterdam; George H. L. Fletcher, Technische Universiteit Eindhoven; Claudio Gutierrez, DCC, Universidad de Chile; Tobias Lindaaker, Neo4j; Marcus Paradies, SAP SE; Stefan Plantikow, Neo4j; Juan Sequeda, Capsenta; Oskar van Rest, Oracle; Hannes Voigt, Technische Universität Dresden.

Manuscript submitted to ACM

LDBC is **not** a standards body and rather than proposing a new standard, we hope that the design and features of G-CORE will guide the evolution of both existing and future graph query languages, towards making them more useful, powerful and expressive.

1 INTRODUCTION

In the last decade there has been increased interest in graph data management. In industry, numerous systems that store and query or analyze such data have been developed. In academia, manifold functionalities for graph databases have been proposed, studied and experimented with.

Graphs are the ultimate abstraction for many real world processes and today the computer infrastructure exists to collect, store and handle them as such. There are several models for representing graphs. Among the most popular is the *property graph data model*, which is a directed graph with labels on both nodes and edges, as well as ⟨property,value⟩ pairs associated with both. It has gained adoption with systems such as AgensGraph [3], Amazon Neptune [4], ArangoDB [9], Blazegraph [12], CosmosDB [24], DataStax Enterprise Graph [14], HANA Graph [37], JanusGraph [21], Neo4j [25], Oracle PGX [38], OrientDB [30], Sparksee [39], Stardog [40], TigerGraph [41], Titan [42], etc. These systems have their own storage models, functionalities, libraries and APIs and many have query languages. This wide range of systems and functionalities poses important interoperability challenges to the graph database industry. In order for the graph database industry to cooperate, community efforts such as Apache Tinkerpop, openCypher[2] and the Linked Data Benchmark Council (LDBC) are providing vendor agnostic graph frameworks, query languages and benchmarks.

LDBC was founded by academia and industry in 2012 [7] in order to establish standard benchmarks for such new graph data management systems. LDBC has since developed a number of graph data management benchmarks [16, 20, 22] to contribute to more objective comparison among systems, informing prospective users of some of the strong- and weak-points of the various systems before even doing a Proof-Of-Concept study, while providing system engineers and architects clear targets for performance testing and improvement. LDBC regularly organizes Technical User Community (TUC) meetings, where not only members report on progress of LDBC task forces but also gather requirements and feedback from data practitioners, who are also present. There have been over 40 graph use-case presentations by data practitioners in these TUC meetings, who often are users of the graph data management software of LDBC members, such as IBM, Neo4j, Ontotext, Oracle and SAP. The topics and contents of these collected TUC presentations show that graph databases are being adopted over a wide range of application fields, as summarized in Figure 1. This further shows that the desired graph query language features are graph pattern matching (e.g., identification of communities in social networks), graph reachability (e.g., fraud detection in financial transactions or insurance), weighted path finding (e.g., route optimization in logistics, or bottleneck detection in telecommunications), graph construction (e.g., data integration in Bioinformatics or specialized publishing domains such as legal) and graph clustering (e.g., on social networks for customer relationship management).

1.1 Three Main Challenges

The following issues are observed about existing graph query languages. These observations are based on the LDBC TUC use-case analysis and feedback from industry practitioners:

Composability. The ability to plug and play is an essential step in standardization. Having the ability to plug outputs and inputs in a query language incentivizes its adoption (modularity, interoperability); simplify abstractions, users do not have to think about multiple data models during the query process; and increases its productivity, by facilitating

Application Fields	
healthcare / pharma	14
publishing	10
finance / insurance	6
cultural heritage	6
e-commerce	5
social media	4
telecommunications	4

Used Features	
graph reachability	36
graph construction	34
pattern matching	32
shortest path search	19
graph clustering	14

Fig. 1. Graph database usage characteristics derived from the use-case presentations in LDBC TUC Meetings 2012-2017 (source: https://github.com/ldbc/tuc_presentations).

reuse and decomposition of queries. Current query languages do not provide full composability because they output tables of values, nodes or edges.

Path as first-class citizens. The notion of Path is fundamental for graph databases, because it introduces an intermediate abstraction level that allows to represents how elements in a graph are related. The facilities provided by a graph query language to manipulate paths (i.e. describe, search, filter, count, annotate, return, etc.) increase the expressivity of the language. Particularly, the ability to return paths enables the user to post-process paths within the query language rather that in an ad-hoc manner [15].

Capture the core of available languages. Both the desirability of a *standard* query language and the difficulty of achieving this, is well-established. This is particularly true for graph data languages due to the diversity of models and the rich properties of the graph model. This motivates our approach to take the successful functionalities of current languages as a base from where to develop the next generation of languages.

1.2 Contributions

Since the lack of a common graph query language kept coming up in LDBC benchmark discussions, it was decided in 2014 to create a task force to work on a common direction for property graph query languages. The authors are members of this task-force.

This paper presents G-CORE, a closed query language on Property Graphs. It is a coherent, comprehensive and consistent integration of industry desiderata and the leading functionalities found in industry practices and theoretical research.
The paper presents the following contributions:

Path Property Graph model. G-CORE treats paths as first-class citizens. This means that paths are outputs of certain queries. The fact that the language must be closed implies that paths must be part of the graph data model. This leads to a principled change of the data model: *it extends property graphs with paths*. That is, in a graph, there is also a (possibly empty) collection of paths; where a path is a concatenation of existing, adjacent, edges. Further, given that nodes, edges and paths are all first-class citizens, paths have identity and can also have labels and ⟨property,value⟩ pairs associated with them. This extended property graph model, called the Path Property Graph model, is backwards-compatible with the property graph model.

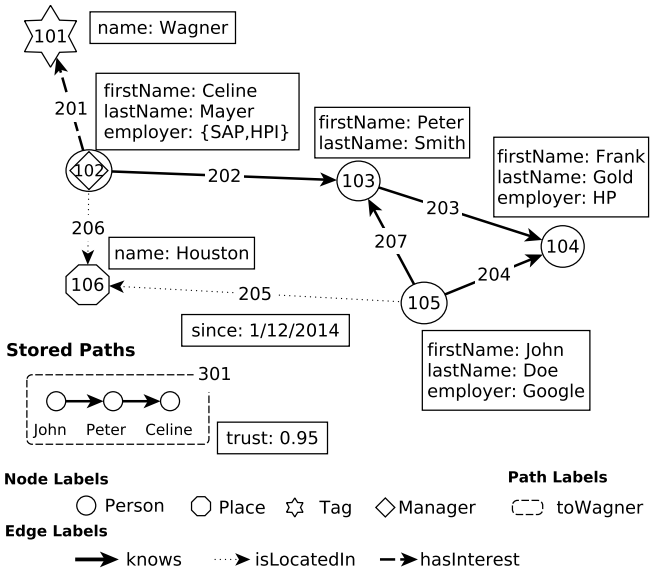

Fig. 2. A small social network. A Path Property Graph (PPG) is a Property Graph that can have "Stored Paths".

Syntax and Semantics of G-CORE. A key contribution is the formal definition of G-CORE. This formal definition prevents any ambiguity about the functionality of the language, thus enabling the development of correct implementations. In particular, an open source grammar for G-CORE is available[1].

Complexity results. To ensure that the query language is practically usable on large data, the design of G-CORE was built on previous complexity results. Features were carefully restricted in such ways that G-CORE is tractable (each query in the language can be evaluated efficiently). Thus, G-CORE provides the most powerful path query functionalities proposed so far, while carefully avoiding intractable complexity.

Organization of the paper. This paper first defines the Extended Property Graph model in Section 2. Then it explains G-CORE in Section 3 via a guided tour, using examples on the LDBC Social Network Benchmark dataset [16], which demonstrate its main features. We summarize our formal contributions, comprising syntax, semantics and complexity analysis of G-CORE in Section 4, while the details of these are described in Appendix A. In Section 5, we show how G-CORE can be easily extended to handle tabular data. We discuss related work in Section 6, before concluding in Section 7.

2 PATH PROPERTY GRAPHS

We first define the data model of G-CORE, which is an extension of the Property Graph data model [6, 25, 27, 35, 44]. We call this model the *Path Property Graph* model, or PPG model for short. Let **L** be an infinite set of label names for nodes, edges and paths, **K** an infinite set of property names, and **V** an infinite set of literals (actual values such as integer

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

and real numbers, strings, dates, truth values $\bot$ and $\top$, that represent true and false, respectively, etc.). Moreover, given a set X, let FSET(X) denote the set of all finite subsets of X (including the empty set), and FLIST(X) denote the set of all finite lists of elements from X (including the empty list).

Definition 2.1. A PPG is a tuple $G = (N, E, P, \rho, \delta, \lambda, \sigma)$, where:

(1) N is a finite set of node identifiers, E is a finite set of edge identifiers and P is a finite set of path identifiers, where N, E and P are pairwise disjoint.
(2) $\rho : E \to (N \times N)$ is a total function.
(3) $\delta : P \to \text{FLIST}(N \cup E)$ is a total function such that for every $p \in P$, it holds that $\delta(p) = [a_1, e_1, a_2, \ldots, a_n, e_n, a_{n+1}]$, where: (i) $n \geq 0$, (ii) $e_j \in E$ for every $j \in \{1, \ldots, n\}$, and (iii) $\rho(e_j) = (a_j, a_{j+1})$ or $\rho(e_j) = (a_{j+1}, a_j)$ for every $j \in \{1, \ldots, n\}$
(4) $\lambda : (N \cup E \cup P) \to \text{FSET}(\mathbf{L})$ is a total function.
(5) $\sigma : (N \cup E \cup P) \times \mathbf{K} \to \text{FSET}(\mathbf{V})$ is a total function for which there exists a finite set of tuples $(x, k) \in (N \cup E \cup P) \times \mathbf{K}$ such that $\sigma(x, k) \neq \emptyset$

Given an edge e in a PPG G, if $\rho(e) = (a, b)$, then a is the starting node of e and b is the ending node of e. The function ρ allows us to have several edges between the same pairs of nodes. Function δ assigns to each path identifier $p \in P$ an actual *path* in G: this is a list $[a_1, e_1, a_2, \ldots, a_n, e_n, a_{n+1}]$ satisfying condition (3) in Definition 2.1. Function λ is used to specify the set of labels of each node, edge, and path, while function σ is used to specify the values of a property for every node, edge, and path. To be precise, if $x \in (N \cup E \cup P)$ and $k \in \mathbf{K}$ is a property name, then $\sigma(x, k)$ is the set of values of the property k for the identifier x. Observe that if $\sigma(x, k) = \emptyset$, then we implicitly assume that property k is not defined for identifier x, as there is no value of this property for this object. Note that although $\mathbf{K}$ is an infinite set of property names, in G only a finite number of properties are assigned values as we assume that there exists a finite set of tuples $(x, k) \in (N \cup E \cup P) \times \mathbf{K}$ such that $\sigma(x, k) \neq \emptyset$.

Example 2.2. As a simple example of a PPG, consider the small social network graph given in Figure 2. Here we have

$$\begin{aligned} N &= \{101, 102, 103, 104, 105, 106\}, \\ E &= \{201, 202, 203, 204, 205, 206, 207\}, \text{ and} \\ P &= \{301\} \end{aligned}$$

as node, edge, and path identifiers, respectively;

$$\begin{aligned} \rho &= \{201 \mapsto (102, 101), \ldots, 207 \mapsto (105, 103)\} \text{ and} \\ \delta &= \{301 \mapsto [105, 207, 103, 202, 102]\} \end{aligned}$$

as edge and path assignments, respectively; and,

$$\lambda = \{101 \mapsto \{\text{Tag}\}, 102 \mapsto \{\text{Person}, \text{Manager}\}, \ldots, 201 \mapsto \{\text{hasInterest}\}, \ldots, 301 \mapsto \{\text{toWagner}\}\}$$

and

$$\sigma = \{(101, \text{name}) \mapsto \{\text{Wagner}\}, \ldots, (205, \text{since}) \mapsto \{1/12/2014\}, \ldots, (301, \text{trust}) \mapsto \{0.95\}\}$$

as label and property value assignments, respectively.

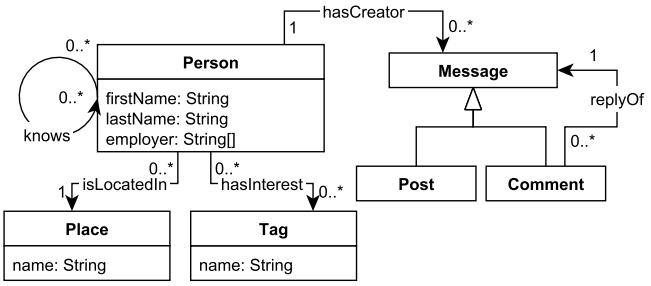

Fig. 3. Social Network Benchmark schema (simplified).

Paths. It is worth remarking that paths are included as a first-class citizens in this data model (at the level of nodes and edges). In particular, paths can have labels and properties, where the latter can be used to describe built-in properties like the length of the path. In our example above, the path with identifier 301 has label "toWagner" and value 0.95 on property "trust".

For convenience, we use $\mathsf{nodes}(p)$ and $\mathsf{edges}(p)$ to denote the list of all nodes and edges of a path bound to a variable p, respectively. Formally, if $\delta(p) = [a_1, e_1, a_2, \ldots, e_n, a_{n+1}]$ then $\mathsf{nodes}(p) = [a_1, \ldots, a_{n+1}]$ and $\mathsf{edges}(p) = [e_1, \ldots, e_n]$. In our example above, $\mathsf{nodes}(301) = [102, 103, 105]$ and $\mathsf{edges}(301) = [202, 207]$.

3 A GUIDED TOUR OF G-CORE

We will now demonstrate and explain the main features of the G-CORE language. The concrete setting is the LDBC Social Network Benchmark (SNB), as illustrated in the simple social network from Figure 2, whose (simplified) schema is depicted in Figure 3. Figure 4 depicts the toy instance (which we refer to as `social_graph`) on which our example queries are evaluated. The use-cases in these examples are data integration and expert finding in a social network.

Always returning a graph. Let us start with what is possibly one of the simplest G-CORE queries:

```
1 CONSTRUCT (n)
2   MATCH   (n:Person)
3     ON    social_graph
4     WHERE n.employer = 'Acme'
```

In G-CORE every query returns a graph, as embodied by the `CONSTRUCT` clause which is at the start of every query body. This example query constructs a new graph with no edges and only nodes, namely those persons who work at Acme – all the labels and properties that these person nodes had in `social_graph` are preserved in the returned result graph.

Match and Filter. The `MATCH..ON..WHERE` clause matches one or more (comma separated) *graph patterns* on a named graph, using the homomorphic semantics [6].

Systems may omit `ON` if there is a *default* graph – let us assume in the sequel that `social_graph` is the default graph. Parenthesis demarcate a node, where `n` is a variable bound to the identity of a node, `:Person` a label, and `n.employer` a property. The G-CORE builds on the ASCII-art syntax from Cypher [17] and the regular path expression syntax from PGQL [43], which has proven intuitive, effective and popular among property graph database users.

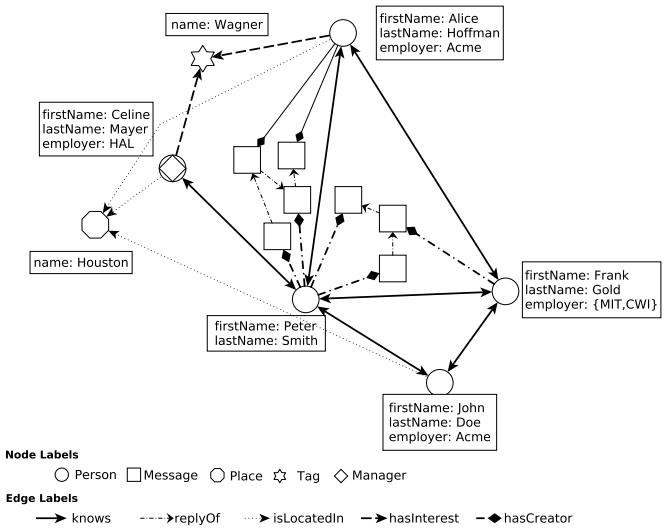

Fig. 4. Initial graph database (social_graph). Note that the knows edges are drawn bi-directionally – this means there are two edges: one in each direction.

The previous example contains a WHERE filter with the obvious semantics: it eliminates all matches where the employer is not Acme.

Multi-Graph Queries and Joins. A more useful query would be a simple *data integration* query, where we might have loaded (unconnected) company nodes into a temporary graph company_graph, but now want to create a unified graph where employees and companies are connected with an edge labeled worksAt. Let us assume that company_graph contains nodes for Acme, HAL, CWI and MIT. As an aside, the real SNB dataset already contains such Company nodes with :worksAt edges to the employees (which in reality do not have an employer property).

The below query has a MATCH clause with two graph patterns, matching these on two different input graphs. Graph patterns that do not have variables in common lead to the Cartesian product of variable bindings, but this query also has a WHERE clause that turns it into an equi-join:

```
5 CONSTRUCT (c)<-[:worksAt]-(n)
6   MATCH   (c:Company) ON company_graph,
7           (n:Person)  ON social_graph
8     WHERE c.name = n.employer
9 UNION social_graph
```

The UNION operator takes it intuitive meaning, and will be touched upon later when we talk about node and edge identity.

Generally speaking, MATCH produces a *set of bindings* which alternatively may be viewed as a table having a column for each variable and one row for each binding. Bindings typically contain node, edge and path identities, whose shape is opaque, but we use intuitive names prefixed # here:

c	n
#Acme	#Alice
#HAL	#Celine
#Acme	#John

Dealing with Multi-Valued properties. In the previous query there is the complication that `n.employer` is *multi-valued* for Frank Gold: he works for both MIT and CWI. Therefore, his person node fails to match with both companies. To explain this, we show the bindings and values of `c.name` and `n.employer` if `WHERE c.name = n.employer` were omitted, and the query would be a Cartesian product:

c	c.name	n	n.employer
#MIT	"MIT"	#Peter	
#CWI	"CWI"	#Peter	
#Acme	"Acme"	#Peter	
#HAL	"HAL"	#Peter	
#MIT	"MIT"	#Frank	{"CWI", "MIT"}
#CWI	"CWI"	#Frank	{"CWI", "MIT"}
#Acme	"Acme"	#Frank	{"CWI", "MIT"}
#HAL	"HAL"	#Frank	{"CWI", "MIT"}
#MIT	"MIT"	#Alice	"Acme"
#CWI	"CWI"	#Alice	"Acme"
#Acme	**"Acme"**	**#Alice**	**"Acme"**
#HAL	"HAL"	#Alice	"Acme"
#MIT	"MIT"	#Celine	"HAL"
#CWI	"CWI"	#Celine	"HAL"
#Acme	"Acme"	#Celine	"HAL"
#HAL	**"HAL"**	**#Celine**	**"HAL"**
#MIT	"MIT"	#John	"Acme"
#CWI	"CWI"	#John	"Acme"
#Acme	**"Acme"**	**#John**	**"Acme"**
#HAL	"HAL"	#John	"Acme"

Notice that according to the definition of our data model, the value of **c.name** is a set. But in the case **c.name** is a singleton set, we omit curly braces, so we simply write `"MIT"` instead of `{"MIT"}`. In the table above, the rows in bold would be the ones that earlier led to bindings surviving the join. Essentially, `"MIT"={"CWI","MIT"}` and `"CWI"={"CWI","MIT"}` evaluate to `FALSE`.

Note that Peter is unemployed, so his `n.employer` value is unbound. More precisely, its Person node does not have an employer property at all. In case of an absent property, its evaluation results in the empty set, which a length test can detect. G-CORE provides `CASE` expressions to coalesce such missing data into other values.

One way to resolve the failing join for Frank, would be to use `IN` instead of `=`, so the comparisons mentioned earlier resolve to `TRUE`:

```
10 CONSTRUCT (c)<-[:worksAt]-(n)
11   MATCH   (c:Company) ON company_graph,
12           (n:Person)  ON social_graph
13     WHERE c.name IN n.employer
14 UNION social_graph
```

Notice that the `IN` operator can be used when `c.name` is a singleton set, as in this case it is natural to ask whether the value in `c.name` is an element of `n.employer`. If we need to compare `c.name` with `n.employer` as sets, then the operator `SUBSET` can be used.

Another way to deal with this in G-CORE is to bind a variable ({`name:e`}) to the `employer` property, which unrolls multi-valued properties into individual bindings:

```
15 CONSTRUCT (c)<-[:worksAt]-(n)
```

```
16   MATCH   (c:Company)             ON company_graph,
17           (n:Person {employer=e}) ON social_graph
18     WHERE c.name = e
19 UNION social_graph
```

Inside the MATCH expression that binds a node, curly braces can be used to bind variables to property values. The set of bindings for this MATCH (which includes the join) now has three variables and the following bindings:

c	n	e
#MIT	#Frank	"MIT"
#CWI	#Frank	"CWI"
#Acme	#Alice	"Acme"
#HAL	#Celine	"HAL"
#Acme	#John	"Acme"

Construction that respects identities. The CONSTRUCT operation fills a graph pattern (used as template) for each binding in the set of bindings produced by the MATCH clause. Edges are denoted with square brackets, and can be pointed towards either direction; in this case there is no edge variable, but there is an edge label :worksAt. Note, to be precise, that CONSTRUCT by default *groups* bindings when creating elements. Nodes are grouped by node identity, and edges by the combination of source and destination node. While five new edges are created here, they are between four existing persons and four existing companies due to this grouping. For instance, the person #Frank, who works for both MIT and CWI, gets two :worksAt edges, to respectively company #MIT and company #CWI.

In the last line of this example query, we UNION-ed these new edges with the original graph, resulting in an enriched graph: the original graph plus five edges. The "full graph" query operators like union and difference are defined in terms of node, edge and path identities. These identities are taken from the input graph(s) of the query. G-CORE is a query language, not an update language. Even though CONSTRUCT allows with SET prop:=val and REMOVE prop to change properties and values (a later example will demonstrate SET), this does not modify the graph database, it just changes the result of that particular query. The practice of returning a graph that shares (parts of) nodes, edges and paths with its inputs, using this concept of identity, provides opportunities for systems to share memory and storage resources between query inputs and outputs.

A shorthand form for the union operation is to include a graph name directly in the comma separated list of CONSTRUCT patterns, as depicted in the next query:

```
20 CONSTRUCT  social_graph,
21     (x GROUP e :Company {name:=e})<-[y:worksAt]-(n)
22   MATCH   (n:Person {employer=e})
```

Graph Aggregation. The above query demonstrates *graph aggregation*. Supposing there would not have been any company nodes in the graph, we might also have created them with this excerpt:

```
CONSTRUCT (n)-[y:worksAt]->(x:Company{name:=n.employer})
```

However, this unbound destination node x would create a company node for *each* binding[2]. This is not what we want: we want only one company per unique name. Graph aggregation therefore allows an explicit GROUP clause in each graph pattern element. Thus, in the above query with GROUP e, we create only one company node for each unique value of e in the binding set. Here the curly brace notation is used inside CONSTRUCT to instantiate the Company.name property in the newly created nodes.

[2]In addition, it would create a company with the name property with the values {"CWI", "MIT"}.

The set of bindings of our graph aggregation query example has the same variables `n` and `e` variables of the previous binding set. The `CONSTRUCT` for node expression `(n)` groups by node identity so instantiates the nodes with identity **#Frank**, **#Alice**, **#Celine** and **#John** in the query result. These nodes were already part of `social_graph`, so given that the `CONSTRUCT` is `UNION`-ed with that, no extra nodes result.

For the `(x GROUP e..)` node expression, `CONSTRUCT` groups by `e` into bindings **"CWI"**, **"MIT"**, **"Acme"**, and **"HAL"** and because `x` is unbound, it will create four new nodes with, say, identities **#CWI**, **#MIT**, **#Acme** and **#HAL**. For the edges to be constructed, G-CORE performs by default grouping of the bindings on the combination of source and destination node, and this results in again five new edges.

When using bound variables in a `CONSTRUCT`, they must be of the right sort: it would be illegal to use `n` (a node) in the place of `y` (an edge) here. In case an *edge* variable (here: `y`) would have been bound (in the `MATCH`), `CONSTRUCT` imposes the restriction that its node variables must also be bound, and be bound to exactly its source and destination nodes, because changing the source and destination of an edge violates its identity. However, it can be useful to bind edges in `MATCH` and use these to construct edges with a new identity, which are copies of these existing edges in terms of labels and property-values. For this purpose, G-CORE supports the `-[=y]-` syntax which makes a copy of the bound `y` edges (as well as the `(=n)` syntax for nodes). Then, the above restriction does not apply. With the copy syntax, it is even possible to copy all labels and properties of a node to an edge (or a path) and vice versa.

In this example, `x` and `y` were unbound and could have been omitted. In the preceding examples, they were in fact omitted. Unbound variables in a `CONSTRUCT` are useful if they occur *multiple* times in the construct patterns, in order to ensure that the same identities will be used (i.e., to connect newly created graph elements, rather than generate independent nodes and edges).

Storing Paths with @p. G-CORE is unique in its treatment of paths, namely as first-class citizens. The below query demonstrates finding the three shortest paths from John Doe towards each other person who lives at his location, reachable over `knows` edges, using Kleene star notation `<:knows*>`:

```
23 CONSTRUCT (n)-/@p:localPeople{distance:=c}/->(m)
24   MATCH   (n)-/3 SHORTEST p<:knows*> COST c/->(m)
25     WHERE (n:Person) AND (m:Person)
26       AND n.firstName = 'John' AND n.lastName = 'Doe'
27       AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
```

In G-CORE, paths are demarcated with slashes `-/ /-`. In the above example `p <:knows*>` binds the shortest path between the single node `n` (i.e. John Doe) and every possible person `m`, under the restriction that this target person lives in the same place. By writing e.g., `-/3 SHORTEST p <:knows*>/->` we obtain multiple shortest paths (at most 3, in this case) for every source–destination combination; if the number 3 would be omitted, it would default to 1. In case there are multiple shortest paths with equal cost between two nodes, G-CORE delivers just any one of them. By writing `p <:knows*> COST c/->` we bind the shortest path cost to variable `c`. By default, the cost of a path is its hop-count (length). We will define weighted shortest paths later. If we would not be interested in the length, `COST c` could be omitted.

In `CONSTRUCT (n)-/@p:localPeople{distance:c}`, we see the bound path variable `@p`. The `@` prefix indicates a **stored path**, that is, this query is delivering a graph with paths. Each path is stored attaching the label `:localPeople`, and its cost as property `distance`.

The graph returned by this query – which lacks a `UNION` with the original `social_graph` – is a *projection* of all nodes and edges involved in these stored paths. We omitted a figure of this for brevity.

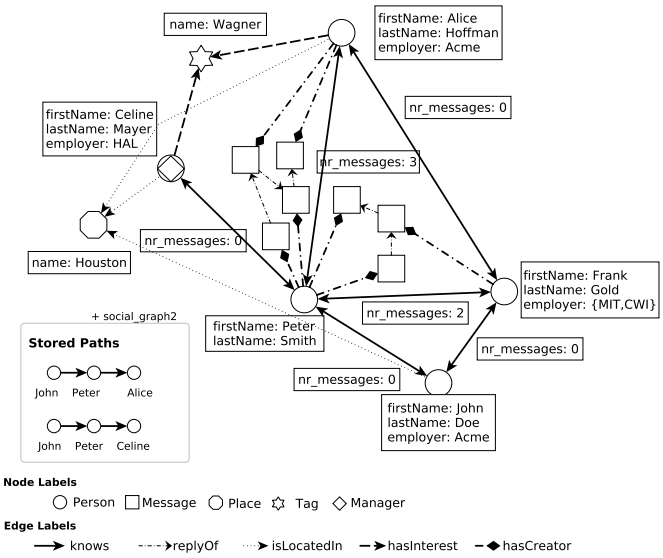

Fig. 5. Graph view social_graph1, which adds nr_message properties to the original social_graph (social_graph2 is social_graph1 plus the Stored Paths in the grey box).

Reachability and All Paths. In a similar query where we just return m, and do not store paths, the <:knows*> path expression semantics is a *reachability* test:

```
28 CONSTRUCT (m)
29   MATCH   (n:Person)-/<:knows*>/->(m:Person)
30     WHERE n.firstName = 'John' AND n.lastName = 'Doe'
31       AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
```

In this case we use -/<:knows*>/-> without the SHORTEST keyword. Using ALL instead of SHORTEST: asking for *all* paths, is not allowed if a path variable is bound to it and used somehwere, as this would be intractable or impossible due to an infinite amount of results. However, G-CORE can support it in the case where the path variable is only used to return a graph projection of all paths:

```
32 CONSTRUCT (n)-/p/->(m)
33   MATCH   (n:Person)-/ALL p<:knows*>/->(m:Person)
34     WHERE n.firstName = 'John' AND n.lastName = 'Doe'
35       AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
```

The method [10] shows how the materialization of all paths can be avoided by summarizing these paths in a graph projection; hence this functionality is tractable.

Existential Subqueries. In the SNB graph, isLocatedIn is not a simple string attribute, but an edge to a city, and the three previous query examples used pattern matching directly in the WHERE clause: (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m). G-CORE allows this and uses implicit existential quantification, which here is equivalent to:

```
36 WHERE EXISTS (
37   CONSTRUCT ()
```

```
38     MATCH (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m) )
```

This constructs one new node (unbound anonymous node variable `()`) for each match of `n` and `m` coinciding in the city where they are located – where that city is represented by a `()` again. Whenever such a subquery evaluates to the empty graph, the automatic existential semantics of `WHERE` evaluates to `FALSE`; otherwise to `TRUE`.

Views and Optionals. The fact that G-CORE is *closed* on the PPG data model means that subqueries and views are possible. In the following example we create such a view:

```
39 GRAPH VIEW social_graph1 AS (
40   CONSTRUCT social_graph,
41         (n)-[e]->(m) SET e.nr_messages := COUNT(*)
42     MATCH (n)-[e:knows]->(m)
43       WHERE (n:Person) AND (m:Person)
44       OPTIONAL (n)<-[c1]-(msg1:Post|Comment),
45                (msg1)-[:reply_of]-(msg2),
46                (msg2:Post|Comment)-[c2]->(m)
47         WHERE (c1:has_creator) AND (c2:has_creator) )
```

The result of this graph view can be seen in Figure 5. To each `:knows` edge, this view adds a `nr_messages` property, using the `SET .. :=` sub-clause. This sub-clause of `CONSTRUCT` can be used to modify properties of nodes, edges and paths that are being constructed. This particular `nr_messages` property contains the amount of messages that the two persons `n` and `m` have actually exchanged, and is a reliable indicator of the intensity of the bond between two persons.

The edge construction `(n)-[e]->(m)` adds nothing new, but as described before, performs implicit graph aggregation, where bindings are grouped on `n`,`m`,`e`, and `COUNT(*)` evaluates to the amount of occurrences of each combination.

This example also demonstrates `OPTIONAL` matches, such that people who know each other but never exchanged a message still get a property `e.nr_messages=0`. All patterns separated by comma in an `OPTIONAL` block must match. Technically, the set of bindings from the main `MATCH` is left outer-joined with the one coming out of the `OPTIONAL` block (and there may be more than one `OPTIONAL` blocks, in which case this repeats). There can be multiple `OPTIONAL` blocks, and each `OPTIONAL` block can have its own `WHERE`; we demonstrate this here by moving some label tests to `WHERE` clauses (on `:Person` and `:has_creator`). This query also demonstrates the use of disjunctive label tests (`msg1:Post|Comment`).

If a query contains multiple `OPTIONAL` blocks, they have to be evaluated from the top to the bottom. For example, to evaluate the following pattern:

```
48 MATCH (n:Person)
49    OPTIONAL (n)-[:worksAt]->(c)
50    OPTIONAL (n)-[:livesIn]->(a)
```

we need to perform the following steps: evaluate `(n:Person)` to generate a binding set T_1, evaluate `(n)-[:worksAt]->(c)` to generate a binding set T_2, compute the left-outer join of T_1 with T_2 to generate a binding set T_3, evaluate `(n)-[:livesIn]->(a)` to generate a binding set T_4, and compute the left-outer join of T_3 with T_4 to generate a binding set T that is the result of evaluating the entire pattern. Obviously, in this case the order of evaluation is not relevant, and the previous pattern is equivalent to:

```
51 MATCH (n:Person)
52    OPTIONAL (n)-[:livesIn]->(a)
53    OPTIONAL (n)-[:worksAt]->(c)
```

However, the order of evaluation can be relevant if the optional blocks of a pattern shared some variables that are not mentioned in the first pattern. For example, in the following expression the variable `a` is mentioned in the optional blocks but not in the first pattern `(n:Person)`:

```
MATCH (n:Person)
  OPTIONAL (n)-[:worksAt]->(a)
  OPTIONAL (n)-[:livesIn]->(a)
```

Arguably, such a pattern is not natural, and it should not be allowed in practice. By imposing the simple syntactic restriction that variables shared by optional blocks have to be present in their enclosing pattern, one can ensure that the semantics of a pattern with multiple `OPTIONAL` blocks is independent of the evaluation order [31].

Weighted Shortest Paths. The finale of this section describes an example of *expert finding*: let us suppose that John Doe wants to go to a Wagner Opera, but none of his friends likes Wagner. He thus wants to know which friend to ask to introduce him to a true Wagner lover who lives in his city (or to someone who can recursively introduce him). To optimize his chances for success, he prefers to try "friends" who actually communicate with each other. Therefore we look for the *weighted* shortest path over the `wKnows` ("weighted knows") *path pattern* towards people who like Wagner, where the weight is the inverse of the number of messages exchanged: the more messages exchanged, the lower the cost (though we add one to the divisor to avoid overflow). For each Wagner lover, we want a shortest path.

In G-CORE, weighted shortest paths are specified over *basic path patterns*, defined by a `PATH .. WHERE .. COST` clause, because this allows to specify a cost value for each traversed path pattern. The specified cost must be numerical, and larger than zero (otherwise a run-time error will be raised), where the full cost of a path (to be minimized) is the sum of the costs of all path segments. If the `COST` is omitted, it defaults to 1 (hop count).

```
GRAPH VIEW social_graph2 AS (
PATH wKnows = (x)-[e:knows]->(y)
  WHERE NOT 'Acme' IN y.employer
  COST 1 / (1 + e.nr_messages)
CONSTRUCT social_graph1, (n)-/@p:toWagner/->(m)
  MATCH   (n:Person)-/p<~wKnows*>/->(m:Person)
    ON    social_graph1
    WHERE (m)-[:hasInterest]->(:Tag {name='Wagner'})
      AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
      AND n.firstName = 'John' AND n.lastName = 'Doe')
```

The result of this graph view (`social_graph2`) was already depicted in Figure 5: it adds to `social_graph1` two stored paths. Apart from `GRAPH VIEW name AS (query)`, and similar to CREATE VIEW in SQL which introduces a global name for a query expression, G-CORE also supports a `GRAPH name AS (query1) query2` clause which, similar to WITH in SQL, introduces a name that is only visible inside `query2`.

Powerful Path Patterns. Basic `PATH` patterns are a powerful building block that allow complex path expressions as concatenations of these patterns [43] using a Kleene star, yet still allow for fast Dijkstra-based evaluation. In G-CORE, these path patterns can even be non-linear shapes, as `PATH` can take a comma-separated list of multiple graph patterns. But, the path pattern must contain a start and end node (a *path segment*), which is taken to be the first and last node in its first graph pattern. This ensures path patterns can be stitched together to form paths – a path pattern always contains a path segment between its start and end nodes. These basic path patterns can also contain `WHERE` conditions,

Matching	
Matching all patterns (Homomorphism)	*
Matching literal values	18, 22
Matching k shortest paths	24
Matching all shortest paths	29
Matching weighted shortest paths	60
(multi-segment) optional matching	44
Querying	
Querying multiple graphs	6
Queries on paths	69
Filtering matches	4,8,13,18,26,30,34,59,64,71
Filtering path expressions	58
Value joins	8
Cartesian product	11
List membership	13
Subqueries	
Set operations on graphs	8, 14, 19
Existential subqueries	
- Implicit	27, 31, 35
- Explicit	36
Construction	
Graph construction	*
Graph aggregation	21
Graph projection	23
Graph views	39, 57
Property addition	41

Table 1. Overview of G-CORE features and their line occurrences in the example queries in Section 3.

without restrictions on their complexity. As John Doe wants his preference for Wagner to remain unknown at his work, we exclude employees of Acme from occurring on the path[3]. The result of this query is a view `social_graph2` in which all these shortest paths from John Doe to Wagner lovers have been materialized (the `@p` in `(n)-/@p:toWagner/->(m)`).

A unique capability of G-CORE is to query and analyze databases of potentially many stored paths. We demonstrate this in the final query, where we score John's friends for their aptitude:

```
67 CONSTRUCT (n)-[e:wagnerFriend {score:=COUNT(*)}]->(m)
68           WHEN e.score > 0
69   MATCH   (n:Person)-/@p:toWagner/->(), (m:Person)
70     ON    social_graph2
71     WHERE n = nodes(p)[1]
```

For the `:toWagner` paths, we use `nodes(p)[1]` to look at the second node in each path, i.e. a direct friend of John Doe. G-CORE starts counting at 0 so `nodes(my_path)[n]` returns the $n-1$ item from the list returned by the function `nodes()`, which returns all nodes on a path. For these direct friends we count how often they occur as the start of `:toWagner` paths. These scores has been attached as a `score` property to new `:wagnerFriend` edges. Since in the toy example there are only two Wagner lovers and thus two shortest paths to them, both via Peter, the result of this query is a single `:wagnerFriend` edge between John and Peter with score 2.

[3]Note that non-linear path patterns, such as `PATH (a)-[]-(b),(b)-(c)` add power over linear patterns with existential filters: `PATH (a)-[]-(b)WHERE (b)-(c)`, because the latter cannot bind variables. In G-CORE, variable `c` can be used in a `COST` expression.

4 FORMALIZING AND ANALYZING G-CORE

One of the main goals of this paper is to provide a formal definition of the syntax and semantics of a graph query language including the features shown in the previous sections. Formally, a G-CORE query is defined by the following top-down grammar:

$$
\begin{aligned}
\textit{query} &::= \textit{headClause fullGraphQuery} \\
\textit{headClause} &::= \varepsilon \mid \textit{pathClause headClause} \mid \textit{graphClause headClause} \\
\textit{fullGraphQuery} &::= \textit{basicGraphQuery} \mid \\
&\qquad (\textit{fullGraphQuery setOp fullGraphQuery}) \\
\textit{setOp} &::= \textsc{Union} \mid \textsc{Intersect} \mid \textsc{Minus} \\
\textit{basicGraphQuery} &::= \textit{constructClause matchClause}
\end{aligned}
$$

Thus, a G-CORE query consists of a sequence of Path and Graph clauses, followed by a full graph query, i.e., a combination of basic graph queries under the set operations of union, intersection and difference. A basic graph query consists of a single Construct clause followed by one Match clause. We have seen examples of all these features in Section 3.

In Appendix A, we provide formal definitions of the syntax and semantics of G-CORE. The basic idea of the language is, given a PPG G, to create a new PPG H using the Construct clause. This is achieved, in turn, by applying an intermediate step provided by the Match clause. The application of such a clause creates a set of bindings Ω, based on a graph pattern that is evaluated over G. The interaction between the Match and the Construct clause is explained in more detail below:

- The result of evaluating the graph pattern φ that defines the content of a Match clause over a PPG G always corresponds to a set Ω of bindings, which is denoted by $[\![\varphi]\!]_G$. The bindings in Ω can then be filtered by using boolean conditions specified in the Where clause.
- A Construct clause ψ then takes as input both the PPG G and the set of bindings Ω, and produces a new PPG H, which is denoted by $[\![\psi]\!]_{\Omega,G}$. Note that G is also an input in the evaluation of ψ, as the set of bindings Ω can make reference to objects whose labels and properties are defined in G.

The role of the Path clause is to define complex path expressions, as well as the cost associated with them, that can in turn be used in graph patterns in the Match clause. In this way, it is possible to define rich navigational patterns on graphs that capture expressive query languages that have been studied in depth in the theoretical community (e.g., the class of *regular queries* [34]).

Complexity analysis. The G-CORE query language has been carefully designed to ensure that G-CORE queries can be evaluated efficiently in *data complexity*. Formally, this means that for each fixed G-CORE query q, the result $[\![q]\!]_G$ of evaluating q over an input PPG G can be computed in polynomial time. The main reasons that explain this fact are given below.

First of all, graph patterns correspond (essentially) to conjunctions of atoms expressing that two nodes are linked by a path satisfying a certain regular expression over the alphabet of node and edge labels. The set Ω of all bindings of a fixed graph pattern φ over the input PPG G can then be easily computed in polynomial time: we simply look for all possible ways of replacing node and edge variables in φ by node and edge identifiers in G, respectively, and then for

each path variable π representing a path in G from node u to node v whose label must conform to a regular expression r, we replace π by the shortest/cheapest path in G from u to v that satisfies r (if it exists). This can be done in polynomial time by applying standard automata-theoretic techniques in conjunction with Dijkstra-style algorithms. (Notice that the latter would not be true if our semantics was based on simple paths; in fact, checking if there is a simple path in an extended property graph whose label satisfies a fixed regular expression is an NP-complete problem [23]).

Suppose, now, that we are given a fixed G-CORE query q that corresponds to a sequence of clauses followed by a full graph query q'. Each clause is defined by a graph pattern φ whose evaluation corresponds to a binary relation over the nodes of the input PPG G. By construction, the graph pattern φ might mention binary patterns which are defined in previous clauses. Therefore, it is possible to iteratively evaluate in polynomial time all graph patterns $\varphi_1, \ldots, \varphi_k$ that are mentioned in the clauses of q. Once this process is finished, we proceed to evaluate q' (which is defined in terms of the φ_i's).

By definition, q' is a boolean combination of full graph queries $q_1, \ldots, q_m$. It is thus sufficient to explain how to evaluate each such a full graph query q_j in polynomial time. We can assume by construction that q_j consists of a Construct clause applied over a Match clause. We first explain how the set of bindings that satisfy the Match clause can be computed in polynomial time. Since one or more Optional clauses could be applied over the Match clause, the semantics is based on the set Ω of *maximal* bindings for the whole expression, i.e., those that satisfy the primary graph pattern expressed in the Match clause, and as many atoms as possible from the basic graph patterns that define the Optional clauses. The computation of Ω can be carried out in polynomial time by a straightforward extension of the aforementioned techniques for efficiently evaluating basic graph patterns. Finally, filtering Ω in accordance with the boolean conditions expressed in the Where clause can easily be done in polynomial time (under the reasonable assumption that such conditions can be evaluated efficiently). Recall that a possible such a condition is Exists Q, for Q a subquery. We then need to check whether the evaluation of Q over G yields an empty graph. We inductively assume the existence of an efficient algorithm for checking this.

Finally, the application of the Construct clause on top G and the set Ω of bindings generated by the Match clause can be carried out in polynomial time. Intuitively, this is because the operations allowed in the Construct clause are defined by applying some simple aggregation and grouping functions on top of bindings generated by relational algebra operations.

Given that all evaluation steps of G-CORE have polynomial complexity in data size, we conclude that G-CORE is tractable.

5 EXTENSIONS OF G-CORE

Practical use of graphs often requires handling tabular data. This suggests that extending G-CORE with additional functionality for projecting tabular results or constructing graphs from imported tabular data may be useful.

Projecting tabular results. In order to integrate the results of graph matching into another system, it would be necessary or at least convenient to be able to produce a tabular projection from a query. It is quite straightforward to imagine the set of bindings produced by `MATCH` as a table, and use that to return a tabular projection. To achieve this, G-CORE could be extended with a `SELECT` clause for projecting expressions into a table. Such a tabular projection clause would also allow the introduction of other common relational operations for slicing, sorting, and aggregation, similar to Cypher's `RETURN` clause or the `SELECT` clauses of SQL or SPARQL. Furthermore, `SELECT` could be used for adding various forms of expression-level subqueries, such as scalar subqueries or list subqueries.

Consider this example of a query that uses tabular projection:

```
SELECT m.lastName + ', ' + m.firstName AS friendName
MATCH (n:Person)-/<:knows*>/->(m:Person)
    WHERE n.firstName = 'John' AND n.lastName = 'Doe'
      AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
```

This query matches persons with the name "John Doe" together with indirect friends that live in the same city and returns a table with the names of these friends.

It should be noted that the introduction of tabular projection into G-CORE changes the language to a multi-sorted language that is capable of either producing a table or a graph. Such a language would no longer be fully closed under graphs in a strict sense, which is one reason why this extension has been left to the future.

Importing tabular data. Conversely, integration of G-CORE with existing systems raises the question of how pre-existing tabular data could be processed in a pure graph query language. Next we present two different alternative proposals for how tabular data could be brought in to the graph world of G-CORE.

Binding table inputs. One way to import tabular data would be through the introduction of a new `FROM <table>` clause that would import sets of scalar bindings from a table, which could be used for defining a graph using the `CONSTRUCT` clause such as in this example:

```
CONSTRUCT
  (cust GROUP custName :Customer {name:=custName}),
  (prod GROUP prodCode :Product {code:=prodCode}),
  (cust)-[:bought]->(prod)
FROM orders
```

This will construct a new graph from an input table of customer names *custName* and product codes *prodCode* by connecting per-customer and per-product nodes as given by the table.

Interpreting tables as graphs. Another alternative is to allow the `MATCH .. ON ..` to treat a tabular input following `ON` as a graph consisting of only isolated nodes that correspond to each row in the table. The properties of these nodes are the columns of the table and the values are the fields of the corresponding row.

If we express the previous example using this syntax, it would now look as follows:

```
CONSTRUCT
  (cust GROUP o.custName :Customer {name:=o.custName}),
  (prod GROUP o.prodCode :Product {code:=o.prodCode}),
  (cust)-[:bought]->(prod)
MATCH (o) ON orders
```

6 DISCUSSION AND RELATED WORK

Graph query languages have been extensively researched in the past decades, and comprehensive surveys are available. Angles and Gutierrez [8] surveyed GQLs proposed during the eighties and nineties, before the emergence of current (practical) graph database systems. Wood [45] studied GQLs focusing on their expressive power and computational complexity. Angles [5] compares graph database systems in terms of their support for essential graph queries. Barceló [11] studies the expressiveness and complexity of several navigational query languages. Recently, Angles et al. [6] presented

a study on fundamental graph querying functionalities (mainly graph patterns and navigational queries) and their implementation in modern graph query languages.

The extensive research on querying graph databases has not give rise yet to a standard query language for property graphs (like SQL for the relational model). Nevertheless, there are several industrial graph database products on the market. Gremlin [18] is a graph-based programming language for property graphs which makes extensive use of XPath to support complex graph traversals. Cypher [13], originally introduced by Neo4j and now implemented by a number of vendors, is a declarative query language for property graphs that has graph patterns and path queries as basic constructs. We primarily consider version 9 of Cypher as outlined by [17, 26], while recognizing that Cypher is an evolving language where several advancements compared to Cypher 9 have already been made. Oracle has developed PGQL [43], a graph query languages that is closely aligned to SQL and that supports powerful regular path expressions. Several implementations of PGQL, both for non-distributed [38] and distributed systems [36], exist. Here, we consider PGQL 1.1 [29], which is the most recent version that is commercially available [28].

G-CORE has been designed to support most of the main and relevant features provided by Cypher, PGQL, and Gremlin. Next we describe the main differences among G-CORE, Cypher, PGQL, and Gremlin based on the query features described in Section 3. Some features (e.g. aggregate operators) will not be discussed here as there are not substantial differences from one language to other.

Graph pattern queries. The notion of basic graph pattern, i.e. the conjunction of node-edge-node patterns with filter conditions over them, is intrinsically supported by Cypher, PGQL and G-CORE. Some differences arise regarding the support for complex graph patterns (i.e. union, difference, optional). Both Cypher and G-CORE define the UNION operator to merge the results of two graph patterns. The absence of graph patterns (negation) is mainly supported via existential subqueries. It is expressed in G-CORE, Cypher and PGQL with the WHERE NOT (EXISTS) clause. Optional graph patterns can be explicitly declared in G-CORE and Cypher with the OPTIONAL clause. PGQL does not support optional graph patterns, although they can be roughly simulated with length-restricted path expressions (see below). Although Gremlin is focused on navigational queries, it supports complex graph patterns (including branches and cycles) as the combination of traversal patterns.

Path queries. G-CORE, Cypher and PGQL support path queries in terms of regular path expressions (i.e. edges can be labeled with regular expressions). The main difference between Cypher 9 and PGQL is that the closure operator is restricted to a single repeated label / value. Both Cypher and PGQL support path length restrictions, a feature that although can be simulated using regular expressions, improves the succinctness of the language. Gremlin supports arbitrary or fixed iteration of any graph traversal (i.e. it is more expressive than regular path queries). Similar to Cypher, Gremlin allows specifying the number of times a traversal should be performed.

Query output. The general approach followed by Cypher 9 and PGQL is to return tables with atomic values (e.g. property values). This approach can be extended such that a result table can contain complex values. The extension in Cypher 9 allows returning nodes, edges, and paths. Recent implementations of Cypher have the ability to return graphs alongside this table [1, 32]. Gremlin also supports returning complete paths as results. In contrast, G-CORE has been designed to return graphs with paths as first class citizens.

Query composition. With the output of a query in G-CORE being a graph, it follows naturally that queries can be composed by querying the output of one query by means of another query. Neither Cypher 9, PGQL or SPARQL supports

this capability. Gremlin supports creating graphs and then populate them before querying the new graph. A notable parallel to G-CORE is the evolution of Cypher 10, where queries are composed through the means of "table-graphs". Cypher 10 expresses queries with multiple graphs and a driving table as input, and produces a set of graphs along with a table as output. This allows Cypher 10 queries to compose both linearly and through correlated subqueries [17].

Evaluation semantics. There are several variations among the languages regarding the semantics for evaluating graph and path expressions. In the context of graph pattern matching semantics, G-CORE, PGQL, and Gremlin follow the homomorphism-based semantics (i.e. no restrictions are imposed during matching), and Cypher 9 follows a no-repeated-edge semantics (i.e. two variables cannot be bound to the same term in a given match) to prevent matching of potentially infinite result sets when enumerating all paths of a pattern. With respect to the evaluation of path expressions, G-CORE uses shortest-path semantics (i.e. paths of minimal length are returned), Cypher 9 implements no-repeated-edge semantics (i.e. each edge occurs at most once in the path), and Gremlin follows arbitrary path semantics (i.e. all paths are considered). Additionally, Cypher 9 and PGQL allow changing the default semantics by using built-in functions (e.g. `allShortestPaths`).

Expressive power versus efficiency. A balance between expressiveness and efficiency (complexity of evaluation) means a balance between practice and theory. Currently no industrial graph query language has a theoretical analysis of its complexity and, conversely, theoretical results have not been systematically translated into a design. One of the main virtues of G-CORE is that its design is the integration of both sources of knowledge and experience.

SPARQL and RDF. In this paper we concentrated on property graphs, but there are other data models and query languages available. A well-known alternative is the Resource Description Framework (RDF), a W3C recommendation that defines a graph-based data model for describing and publishing Web metadata. RDF has a standard query language, SPARQL [33], which was designed to support several types of complex graph patterns (including union and optional). Its latest version, SPARQL 1.1 [19], adds support for negation, regular path queries (called *property paths*), subqueries and aggregate operators. The path queries support reachability tests, but paths cannot be returned, nor can the cost of paths be computed. The evaluation of SPARQL graph patterns follows a homomorphism-based bag semantics, whereas property paths are evaluated using an arbitrary paths semantics [6]. SPARQL allows queries that return RDF graphs, however creating graphs consisting of multiple types of nodes (e.g., belonging to different RDF schema classes; having different properties) in one query is not possible as SPARQL lacks flexible graph aggregation: its CONSTRUCT directly instantiates a single binding table without reshaping. Such constructed RDF graphs can not be reused as subqueries, that is, for composing queries; nor does the language offer "full graph" operations to union or diff at the graph level. We think the ideas outlined in G-CORE could also inspire further development of SPARQL.

7 CONCLUSIONS

Graph databases have come of age. The number of systems, databases and query languages for graphs, both commercial and open source, indicates that these technologies are gaining wide acceptance [3, 4, 9, 12, 14, 21, 24, 25, 30, 37–42].

At this stage, it is relevant to begin making efforts towards interoperability of these systems. A language like G-CORE could work as a base for integrating the manifold models and approaches towards querying graphs.

We defend here two principles we think should be at the foundations of the future graph query languages: *composability*, that is, having graphs and their mental model as departure and ending point and treat the most popular feature of graphs, namely *paths, as first class citizens*.

The language we present, G-CORE, which builds on the experiences with working systems, as well as theoretical results, show these desiderata are not only possible, but computationally feasible and approachable for graph users. This paper is a call to action for the stakeholders driving the graph database industry.

REFERENCES

[1] 2017. Cypher for Apache Spark. (2017). https://github.com/opencypher/cypher-for-apache-spark

[2] 2017. The openCypher Project. (2017). http://www.openCypher.org

[3] AgensGraph - The Performance-Driven Graph Database. 2017. (2017). http://www.agensgraph.com/

[4] Amazon Neptune - Fast, reliable graph database build for cloud. 2017. (2017). https://aws.amazon.com/neptune/

[5] Renzo Angles. 2012. A comparison of current graph database models. In *4rd Int. Workshop on Graph Data Management: Techniques and Applications*.

[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *Comput. Surveys* 50, 5 (2017). https://doi.org/10.1145/3104031

[7] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *SIGMOD Record* 43, 1 (May 2014), 27–31.

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

[9] ArangoDB - Native multimodel database. 2017. (2017). https://arangodb.com/

[10] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *TODS* 37, 4, Article 31 (Dec. 2012), 46 pages.

[11] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proc. of the 32nd Symposium on Principles of Database Systems (PODS)*. ACM, 175–188.

[12] Blazegraph. 2017. (2017). https://www.blazegraph.com/

[13] Cypher - Graph Query Language. 2017. (2017). http://neo4j.com/developer/cypher-query-language/

[14] DataStax Enterprise Graph. 2017. (2017). https://www.datastax.com/products/datastax-enterprise-graph

[15] Anton Dries, Siegfried Nijssen, and Luc De Raedt. 2009. A Query Language for Analyzing Networks. In *Proc. of the 18th ACM Conference on Information and Knowledge Management (CIKM)*. ACM, 485–494.

[16] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD2015*. ACM, 619–630.

[17] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. (2018).

[18] Gremlin - A graph traversal language. 2017. (2017). https://github.com/tinkerpop/gremlin

[19] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language - W3C Recommendation. https://www.w3.org/TR/sparql11-query/. (March 21 2013).

[20] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms. *PVLDB* 9, 13 (Sept. 2016), 1317–1328.

[21] JanusGraph - Distributed graph database. 2017. (2017). http://janusgraph.org/

[22] Venelin Kotsev, Orri Erling, Atanas Kiryakov, Irini Fundulaki, and Vladimir Alexiev. 2017. The Semantic Publishing Benchmark v2.0. (2017). github.com/ldbc/ldbc_spb_bm_2.0/blob/master/doc/LDBC_SPB_v2.0.docx

[23] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.

[24] Microsoft Azure Cosmos DB. 2017. (2017). https://docs.microsoft.com/en-us/azure/cosmos-db/introduction

[25] Neo4j. 2017. The Neo4j Developer Manual v3.3. (2017).

[26] openCypher. 2017. Cypher Query Language Reference, Version 9. (2017). https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf.

[27] The openCypher implementer's group. 2017. Property Graph Model. (2017). https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc

[28] Oracle. 2017. Oracle Big Data Spatial and Graph. (2017). http://www.oracle.com/technetwork/database/database-technologies/bigdata-spatialandgraph/

[29] Oracle. 2017. PGQL 1.1 Specification. (2017). http://pgql-lang.org/spec/1.1/

[30] OrientDB - Multi-Model Database. 2017. (2017). http://orientdb.com/

[31] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[32] Stefan Plantikow, Martin Junghanns, Petra Selmer, and Max Kießling. 2017. Cypher and Spark: Multiple Graphs and More in openCypher. (2017). https://www.youtube.com/watch?v=EaCFxDxhtsI

[33] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF - W3C Recommendation. https://www.w3.org/TR/rdf-sparql-query/. (2008).

[34] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2017. Regular Queries on Graph Databases. *Theory Comput. Syst.* 61, 1 (2017), 31–83.

[35] Marko A. Rodriguez and Peter Neubauer. 2010. Constructions from Dots and Lines. *Bulletin of the American Society for Information Science and Technology* 36, 6 (Aug. 2010), 35–41.
[36] Nicholas P Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. 2017. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. (2017).
[37] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database.. In *BTW*, Vol. 13. 403–420.
[38] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. 2016. Using domain-specific languages for analytic graph databases. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1257–1268.
[39] Sparksee - Scalable high-performance graph database. 2017. (2017). http://www.sparsity-technologies.com/#sparksee
[40] Stardog - The Knowledge Graph Platform for the Enterprise. 2017. (2017). http://www.stardog.com/
[41] TigerGraph - The First Native Parallel Graph. 2017. (2017). https://www.tigergraph.com/
[42] Titan - Distributed Graph Database. 2017. (2017). http://titan.thinkaurelius.com/
[43] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES2016*. ACM, 7.
[44] Hannes Voigt. 2017. Declarative Multidimensional Graph Queries, Patrick Marcel and Esteban Zimányi (Eds.). *Business Intelligence – 6th European Summer School, eBISS 2016, Tours, France, July 3-8, 2016, Tutorial Lectures* 280, 1–37.
[45] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.

A A FORMAL DEFINITION OF G-CORE

In this section we formally present the semantics of G-CORE queries. For ease of presentation, we focus on a simplified syntax equal in expressive power to the full syntax of the language.

A.1 Basic notions

G-CORE queries are recursively defined by the top-down grammar presented in Section 4. In this section we define in detail the different components of the G-CORE grammar. But before doing so, it is important to introduce some basic notions that are used in the definition of their semantics.

Recall the domains used by the PPG data model as defined in Section 2. Let **L** be an infinite set of label names for nodes, edges and paths, **K** an infinite set of property names and **V** an infinite set of literals (i.e. actual values like integers).

Paths conforming to regular expressions. In graph query languages, one is typically interested in checking if two nodes are linked by a path whose label satisfies a regular expression r, and computing one (or more) of such paths if needed. Some graph query languages. e.g., Cypher 9 [17], define the semantics of path expressions based on *simple paths* only (those without repetition of nodes and/or edges), which is known to easily lead to intractability in data complexity [23]. For this reason, in G-CORE we follow a long tradition of graph query languages introduced in the last 30 years and define the semantics of path expressions based on arbitrary paths (see, e.g., [6]). In addition, for the problem of computing a path from node u to node v that satisfies a given regular expression r, we choose to compute the *shortest* such a path according to a fixed lexicographical order on nodes.[4] The reason for this is that checking for the existence of an arbitrary path in a PPG G from u to v that conforms to a regular expression r, and computing the shortest path that witnesses this fact, can be done in polynomial time by applying standard automata techniques in combination with depth-first search. We define the notion of (shortest) paths conforming to regular expressions below.

[4]We acknowledge that using a fixed lexicographical order could be too restrictive when choosing a single path, so a system implementing G-CORE could use a different criterion based, for instance, on whether it can be evaluated more efficiently.

We start by defining the notion of regular expression used in G-CORE. A regular expression r is specified by the grammar:

$$r \quad ::= \quad \_ \mid \ell \mid \ell^- \mid !\ell \mid (r + r) \mid (rr) \mid (r)^*,$$

where $\ell \in \mathbf{L}$. Intuitively, an expression of the form either ℓ or ℓ^-, with $\ell \in \mathbf{L}$, refers to an edge label, while $!\ell$ refers to a node label. The expression _ is used as a wildcard that stands for "any label". Notice that under this definition, the alphabet of every regular expression is a finite subset of $\{\_\} \cup \mathbf{L} \cup \{\ell^- \mid \ell \in \mathbf{L}\} \cup \{!\ell \mid \ell \in \mathbf{L}\}$.

Let $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ be a PPG and $L = [a_1, e_1, a_2, \ldots, a_n, e_n, a_{n+1}]$ be a path over G. Then given a regular expression r, we say that L is a path from a_1 to a_{n+1} conforming to r if there exists a string $u_1 v_2 u_2 \ldots u_n v_{n+1}, u_{n+1}$ in the regular language defined by r such that:

- For each $i \in \{1, \ldots, n+1\}$, either $u_i = \_$ or $u_i = !\ell$ for some $\ell \in \lambda(a_i)$.
- For each $j \in \{1, \ldots, n\}$, either
  - $v_j = \_$, or
  - $\rho(e_j) = (a_j, a_{j+1})$ and $v_j = \ell$ for some $\ell \in \lambda(e_j)$, or
  - $\rho(e_j) = (a_{j+1}, a_j)$ and $v_j = \ell^-$ for some $\ell \in \lambda(e_j)$.

The length of a path $L = [a_1, e_1, a_2, \ldots, a_n, e_n, a_{n+1}]$, written length($L$), is n. Then L is a *shortest path* from a node a to a node b conforming to a regular expression r, if for every path L' from a to b that conforms to r, it holds that length(L) $\leq$ length(L').

Bindings. From now on, we assume that $\mathcal{N}$, $\mathcal{E}$ and $\mathcal{P}$ are countably infinite sets of node, edge and path variables, respectively, which are pairwise disjoint. Let $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ be a PPG. A *binding* μ over G is a partial function $\mu : (\mathcal{N} \cup \mathcal{E} \cup \mathcal{P}) \to (N \cup E \cup P)$ such that $\mu(x) \in N$ if $x \in \mathcal{N}$, $\mu(y) \in E$ if $y \in \mathcal{E}$, and $\mu(z) \in P$ if $z \in \mathcal{P}$. The domain of a binding μ is denoted by dom(μ), and it is assumed to be finite. Two bindings μ_1 and μ_2 are said to be *compatible*, denoted by $\mu_1 \sim \mu_2$, if for every variable $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it holds that $\mu_1(x) = \mu_2(x)$. Notice that if μ_1 and μ_2 are compatible bindings, then $\mu_1 \cup \mu_2$ is a well-defined function.

Let Ω_1, Ω_2 be finite sets of bindings over G. The following four basic operations will be extensively used in this article:

$$\begin{aligned}
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\},\\
\Omega_1 \ltimes \Omega_2 &= \{\mu_1 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\},\\
\Omega_1 \setminus \Omega_2 &= \{\mu_1 \mid \mu_1 \in \Omega_1 \text{ and } \nexists \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\},\\
\Omega_1 \bowtie\!\!\!\!\!\!\subset \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2).
\end{aligned}$$

Given a countably infinite set of value variables $\mathcal{V}$ disjoint from $\mathcal{N}$, $\mathcal{E}$ and $\mathcal{P}$, it is straightforward to extend our definition of bindings (and our presentation of the semantics of G-CORE) to capture assignments of literals to variables, i.e., such an extended binding would be a partial function $\mu : (\mathcal{N} \cup \mathcal{E} \cup \mathcal{P} \cup \mathcal{V}) \to (N \cup E \cup P \cup \mathbf{V})$ such that $\mu(x) \in \mathbf{V}$ if $x \in \mathcal{V}$. For ease of presentation, however, we do not further consider such extended bindings in the sequel.

Expressions. We next define expressions ξ according to the following grammar:

$$\xi ::= x \mid x.k \mid x{:}\ell \mid \diamond\xi \mid \xi \odot \xi \mid f(\xi, \xi, \ldots) \mid \Sigma(\xi) \mid \textsc{Exists}\ q,$$

where x is a variable, $k \in \mathbf{K}$ is property key, $\ell \in \mathbf{L}$ is a label, $\diamond$ is a unary operator, $\odot$ is a binary operator, f is a built-in function whose value only depends on its inputs and Σ is an aggregation function. Unary operators are boolean negation Not, arithmetic negation -1, etc. Binary operators are comparisons such as $=, \neq, <, \leq$, Subset Of, In, . . ., boolean operations such as And and Or, arithmetic operations on numbers and strings such as $+, -, *, /, \ldots$, etc. Built-in functions include the standard ones for type casting, string, date and collection handling (e.g. Size) known from other query languages, as well as graph-specific function such as Labels, which returns the set of labels, and Nodes, and Edges, which return the list of nodes and edges of o, respectively, when applied to an object identifier o. Aggregation functions include the standard ones inherited from relational query languages, Count, Min, Max, Sum, Avg, etc. and Collect to collect all values of the group into a collection.

The semantics of expressions inherited from other query languages is defined in analogy with them, so we do not repeat it here. We denote the evaluation of any such an expression ξ over a PPG $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ and a set of bindings Ω as $[\![\xi]\!]_{\Omega,G}$. In any such a case, $[\![\xi]\!]_{\Omega,G} \in N \cup E \cup P \cup \mathbf{L} \cup \mathbf{V}$ (recall that $\mathbf{V}$ includes truth values $\bot$ and $\top$). The semantics of the G-CORE specific expressions, on the other hand, is defined as follows (where we denote by μ the singleton binding set $\{\mu\}$):

- If ξ is a variable x, then $[\![\xi]\!]_{\mu,G} = [\![x]\!]_{\mu,G} = \mu(x)$.
- If $\xi = x.k$, then $[\![\xi]\!]_{\mu,G} = \sigma([\![x]\!]_{\mu,G}, k) = \sigma(\mu(x), k)$.
- If $\xi = x{:}\ell$, then $[\![\xi]\!]_{\mu,G} = \top$ iff $\ell \in \lambda([\![x]\!]_{\mu,G})$ iff $\ell \in \lambda(\mu(x))$.
- If $\xi = \textsc{Labels}(o)$, then $[\![\xi]\!]_{\mu,G} = \lambda([\![o]\!]_{\mu,G})$.
- If $\xi = \textsc{Nodes}(o)$, then $[\![\xi]\!]_{\mu,G} = \mathsf{nodes}([\![o]\!]_{\mu,G})$.
- If $\xi = \textsc{Edges}(o)$, then $[\![\xi]\!]_{\mu,G} = \mathsf{edges}([\![o]\!]_{\mu,G})$.
- If $\xi = \textsc{Exists}\ q$, then $[\![\xi]\!]_{\mu,G} = \top$ iff $(N \neq \emptyset)$, assuming that $[\![q]\!]_{\mu,G} = (N, E, P, \rho, \delta, \lambda, \sigma)$.

A.2 The MATCH clause

Basic graph patterns. A *basic graph pattern* is specified by the following grammar:

$$\begin{aligned} \mathit{basicGraphPattern} &::= \mathit{nodePattern} \mid \mathit{edgePattern} \mid \mathit{pathPattern} \\ \mathit{nodePattern} &::= (x) \\ \mathit{edgePattern} &::= x \xrightarrow{z} y \\ \mathit{pathPattern} &::= x \xRightarrow{@w \text{ in } r} y \mid x \xRightarrow{w \text{ in } r} y \end{aligned}$$

where $x, y \in \mathcal{N}$, $z \in \mathcal{E}$, $w \in \mathcal{P}$, and r is a regular expression.

Let α be a basic graph pattern. The evaluation of α over a PPG $G = (N, E, P, \rho, \delta, \lambda, \sigma)$, denoted by $[\![\alpha]\!]_G$, is inductively defined:

- If α is a node pattern (x), then $[\![\alpha]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \{x\} \text{ and } \mu(x) \in N\}$.
- If α is an edge pattern $x \xrightarrow{z} y$, then $[\![\alpha]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \{x, y, z\},\ \mu(x), \mu(y) \in N, \mu(z) \in E \text{ and } \rho(\mu(z)) = (\mu(x), \mu(y))\}$.

- If α is a path pattern $x \xrightarrow{@w \text{ in } r} y$, then $[\![\alpha]\!]_G = \{\mu \mid \text{dom}(\mu) = \{x, y, w\}, \mu(x), \mu(y) \in N, \mu(w) \in P \text{ and } \delta(\mu(w))$ is the shortest path from $\mu(x)$ to $\mu(y)$ that conforms to $r\}$.
- If α is a path pattern $x \xrightarrow{w \text{ in } r} y$, then $[\![\alpha]\!]_G = \{\mu \mid \text{dom}(\mu) = \{x, y, w\}, \mu(x), \mu(y) \in N$, and $\mu(w)$ is a fresh path identifier (i.e., $\mu(w) \notin P$) associated to the shortest path L from $\mu(x)$ to $\mu(y)$ over G that conforms to $r\}$.

Basic graph patterns with location and full graph patterns. Basic graph patterns can be evaluated over different graphs in G-CORE. More specifically, we use an On operator to specify the location where a basic graph pattern has to be evaluated:

$$\begin{aligned} \textit{basicGraphPatternLocation} ::= \textit{basicGraphPattern} \mid \\ \textit{basicGraphPattern} \text{ On } \textit{location} \\ \textit{location} ::= \textit{gid} \mid \textit{fullGraphQuery}, \end{aligned}$$

where *gid* is a graph identifier and *fullGraphQuery* is as defined in the G-CORE grammar given in Section 4. For reasons that will become clear later, we can always assume the evaluation $[\![\alpha]\!]_G$ of a full graph query α over a PPG G to be another PPG H.

To define the semantics of basic graph patterns with location, we assume given a function gr that associates an actual graph, denoted by gr(*gid*), to every graph identifier *gid*. Then given a basic graph pattern with location β, the evaluation of β over a PPG G, denoted by $[\![\beta]\!]_G$, is defined as follows.

- If β is α On *gid*, where α is a basic graph pattern and *gid* is a graph identifier, then $[\![\beta]\!]_G = [\![\alpha]\!]_{\text{gr}(gid)}$.
- If β is α On Q, where α is a basic graph pattern, Q is a full graph query, and $H = [\![Q]\!]_G$, then $[\![\beta]\!]_G = [\![\alpha]\!]_H$.

Finally, a full graph pattern is defined as a sequence of basic graph patterns with locations separated by comma:

$$\begin{aligned} \textit{fullGraphPattern} ::= \textit{basicGraphPatternLocation} \mid \\ \textit{basicGraphPatternLocation}, \ \textit{fullGraphPattern} \end{aligned}$$

To define the evaluation of such pattern γ over a PPG G, denoted by $[\![\gamma]\!]_G$, we only need to consider the following recursive rule:

- If γ is a β, γ', where β is a basic graph pattern with location and γ' is a full graph pattern, then $[\![\gamma]\!]_G = [\![\beta]\!]_G \bowtie [\![\gamma']\!]_G$.

Conditions on binding tables. G-CORE includes a Where clause that allows to filter bindings according to some boolean *conditions*. It is important to mention that this clause cannot be used to generate new bindings, it can only be used to filter the bindings that are produced by a Match clause.

A Where clause can include conditions of the form Exists Q, where Q is a sub-query. Such a query evaluates to true if the evaluation of Q is a non-empty PPG, and it evaluates to false otherwise. It is important to consider that Q can share some variables with the outer query, which are considered as parameters as in the case of correlated subqueries in SQL. Thus, in order to define the semantics of Exists Q, it is necessary to extend the definition of the semantics of full graph patterns to consider some previously computed bindings. More precisely, given a full graph pattern γ and a set Ω of bindings, the evaluation of γ over G given Ω, denoted by $[\![\gamma]\!]_{\Omega,G}$, is defined as $[\![\gamma]\!]_{\Omega,G} = [\![\gamma]\!]_G \ltimes \Omega$. Notice that if $\Omega = \{\mu_\emptyset\}$, where $\mu_\emptyset$ is the binding with empty domain, then $[\![\gamma]\!]_{\Omega,G} = [\![\gamma]\!]_G$.

We now have the necessary ingredients to define the semantics of MATCH clauses with conditions. The syntax of such expressions is given by the following grammar:

$$\begin{aligned} \textit{matchClause} ::= {} & \textsc{Match}\ \textit{fullGraphPattern}\ \mid \\ & \textsc{Match}\ \textit{fullGraphPattern}\ \textsc{Where}\ \textit{BooleanCondition}, \end{aligned}$$

where *BooleanCondition* is a Boolean-valued expression, that is, an expression that evaluates to $\top$ or $\bot$ (expressions are formally defined in Section A.1). To define the evaluation of a MATCH clause over a PPG G given a set of Ω of bindings, we only need to consider the following rule:

- If γ is a full graph pattern and ξ is a Boolean-valued expression, then $[\![\textsc{Match}\ \gamma\ \textsc{Where}\ \xi]\!]_{\Omega,G} = \{\mu \in [\![\gamma]\!]_{\Omega,G} \mid [\![\xi]\!]_{\mu,G} = \top\}$.

Optional full graph patterns. In its most general version, full graph patterns can also use an OPTIONAL operator to include optional information if present. Thus, the general syntax of MATCH clauses is specified by the following grammar:

$$\begin{aligned} \textit{fullGraphPatternCondition} ::= {} & \textit{fullGraphPattern}\ \mid \\ & \textit{fullGraphPattern}\ \textsc{Where}\ \textit{BooleanCondition} \\ \textit{matchClause} ::= {} & \textsc{Match}\ \textit{fullGraphPatternCondition}\ \mid \\ & \textsc{Match}\ \textit{fullGraphPatternCondition}\ \textit{optionalClause} \end{aligned}$$

where *optionalClause* is given by the following grammar:

$$\begin{aligned} \textit{optionalClause} ::= {} & \textsc{Optional}\ \textit{fullGraphPatternCondition}\ \mid \\ & \textsc{Optional}\ \textit{fullGraphPatternCondition}\ \textit{optionalClause} \end{aligned}$$

The semantics of the OPTIONAL operator is defined by using the operator $\bowtie\!\!\!\!\subset$ on sets of bindings. More precisely, given a sequence of full graph patterns $\gamma, \gamma_1, \ldots, \gamma_k$, where $k \geq 1$, and a sequence of Boolean conditions $\xi, \xi_1, \ldots, \xi_k$, we have that:

$$\begin{aligned} & [\![\textsc{Match}\ \gamma\ \textsc{Where}\ \xi \\ & \textsc{Optional}\ \gamma_1\ \textsc{Where}\ \xi_1 \cdots \textsc{Optional}\ \gamma_k\ \textsc{Where}\ \xi_k]\!]_{G,\Omega} = \\ & ((\cdots(([\![\textsc{Match}\ \gamma\ \textsc{Where}\ \xi]\!]_{G,\Omega} \bowtie\!\!\!\!\subset [\![\textsc{Match}\ \gamma_1\ \textsc{Where}\ \xi_1]\!]_{G,\Omega}) \\ & \qquad \bowtie\!\!\!\!\subset \cdots)\cdots) \bowtie\!\!\!\!\subset [\![\textsc{Match}\ \gamma_k\ \textsc{Where}\ \xi_k]\!]_{G,\Omega}). \end{aligned}$$

Example. Consider MATCH γ WHERE ξ where

$$\gamma = x \xrightarrow{\text{locatedIn}} w,\ y \xrightarrow{\text{locatedIn}} w,\ x \xRightarrow{@z \text{ in } (\text{knows}+\text{knows}^-)^*} y$$

$$\xi = w.\text{name} = \textit{Houston}.$$

On the PPG $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ given in Figure 2 and formalized in Section 2, we have that

$$\begin{aligned}
[\![\gamma]\!]_G &= [\![x \xrightarrow{\text{locatedIn}} w]\!]_G \bowtie [\![y \xrightarrow{\text{locatedIn}} w]\!]_G \\
&\quad \bowtie [\![x \xRightarrow{@z \text{ in } (\text{knows}+\text{knows}^-)^*} y]\!]_G \\
&= \{\{x \mapsto 105, w \mapsto 106\}, \{x \mapsto 102, w \mapsto 106\}\} \\
&\quad \bowtie \{\{y \mapsto 102, w \mapsto 106\}, \{y \mapsto 105, w \mapsto 106\}\} \\
&\quad \bowtie \{\{z \mapsto 301, x \mapsto 105, y \mapsto 102\}\} \\
&= \{\{x \mapsto 105, y \mapsto 102, w \mapsto 106\}, \{x \mapsto 105, y \mapsto 105, w \mapsto 106\}, \\
&\quad \{x \mapsto 102, y \mapsto 102, w \mapsto 106\}, \{x \mapsto 102, y \mapsto 105, w \mapsto 106\}\} \\
&\quad \bowtie \{\{z \mapsto 301, x \mapsto 105, y \mapsto 102\}\} \\
&= \{\{x \mapsto 105, y \mapsto 102, w \mapsto 106, z \mapsto 301\}\}
\end{aligned}$$

and, letting $\mu = \{x \mapsto 105, y \mapsto 102, w \mapsto 106, z \mapsto 301\} \in [\![\gamma]\!]_G$, we then have that

$$[\![\xi]\!]_{\mu,G} = [\![w.\text{name} = \mathit{Houston}]\!]_{\mu,G} = \top.$$

Therefore, $[\![\textsc{Match}\ \gamma\ \textsc{Where}\ \xi]\!]_G = \{\{x \mapsto 105, y \mapsto 102, w \mapsto 106, z \mapsto 301\}\}$.

A.3 The CONSTRUCT clause

The definition of the CONSTRUCT clause assumes finite set of variables $B \subset \mathcal{N} \cup \mathcal{E} \cup \mathcal{P} \cup \mathcal{V}$. The set contains the variables declared in all full graph patterns of the MATCH clause. B is a syntactical property of a given query. We use $G_\emptyset$ to denote an empty graph.

Syntax of Basic Construct Patterns. The *basic constructs* are specified by the following grammar:

$$\begin{aligned}
\mathit{basicConstruct} &::= \mathit{objectConstructsList} \mid \\
&\qquad \mathit{objectConstructsList}\ \textsc{When}\ \xi \\
\mathit{objectConstructsList} &::= \mathit{gid} \mid \mathit{objectConstruct} \mid \\
&\qquad \mathit{objectConstruct}\ \mathit{objectConstructsList}
\end{aligned}$$

where *gid* is a graph identifier.

Object constructs with respect to the set of variables B are specified by the following grammar:

$$\begin{aligned}
\mathit{objectConstruct} &::= \mathit{nodeContruct} \mid \mathit{relationshipContruct} \\
\mathit{relationshipContruct} &::= \mathit{edgeContruct} \mid \mathit{pathContruct} \\
\mathit{nodeContruct} &::= (x\ \textsc{Group}\ \Gamma; S) \\
\mathit{edgeContruct} &::= x \xrightarrow{z\ \textsc{Group}\ \Gamma; S} y \\
\mathit{pathContruct} &::= x \xRightarrow{@u; S} y
\end{aligned}$$

where $x, y \in \mathcal{N}$, $z \in \mathcal{E}$, $u \in \mathcal{P}$, $\Gamma \subseteq B$ is a grouping sets, S is a set of assignments, and o is the optional symbol @ (i.e., can appear or can be left out). Each object construct pattern has a *construct variable*, which is unique within the basic construct. For a node construct (x GROUP $\Gamma; S$), an edge construct $x \xrightarrow{z \text{ GROUP } \Gamma;S} y$, and a path construct $x \xRightarrow{@u;S} y$, the construct variable is x, z, and u, respectively. In situation of ambiguity, we refer to Γ and S of a specific object construct g by subscripting Γ and S with g's construct variable.

We further constrain the syntax of a basic construct b used in the context of a basic graph query, as follows.

- In each node construct appearing in b, if $x \in B$ then $\Gamma = \{x\}$.
- In each edge construct $x \xrightarrow{z \text{ GROUP } \Gamma z;S_z} y$ appearing in b, it holds that
  - node construct (x GROUP $\Gamma x; S_x$) and (y GROUP $\Gamma y; S_y$) must appearing in b;
  - $\Gamma x \cup \Gamma y \cup \{x, y\} \subseteq \Gamma z$; and
  - if $z \in B$, then $x, y \in B$.

  Furthermore, if $z \in B$, then $\Gamma z = \{z\} \cup \Gamma x \cup \Gamma y$.
- In each path construct $x \xRightarrow{ou;S} y$ appearing in b, it must hold that a path pattern $x \xRightarrow{@u \text{ in } r} y$ appears in the match clause, i.e. $x, y, u \in B$. By definition the grouping set for a path construct is $\Gamma = \{x, y, u\}$.

The conditions ensure, that all object constructs in a basic construct form a valid directed graph and, hence, the use of the WHEN sub-clause can not cause the creation of dangling edges in the resulting PPG.

Semantics of Basic Construct Patterns. The semantics of any basic construct b with respect to a set of bindings Ω evaluated on G, is a pair $\left([\![b]\!]^{\text{graph}}_{\Omega,G}, [\![b]\!]^{\text{bindings}}_{\Omega,G}\right)$, where the $[\![b]\!]^{\text{graph}}_{\Omega,G}$ is a PPG and $[\![b]\!]^{\text{bindings}}_{\Omega,G}$ is a set of bindings. In case where b is a graph identifier *gid*, the PPG resulting from the basic construct pattern is the actual graph associated with *gid*, i.e. $[\![gid]\!]^{\text{graph}}_{\Omega,G} = \text{gr}(gid)$ and $[\![gid]\!]^{\text{bindings}}_{\Omega,G} = \{\mu_\emptyset\}$.

For the remaining cases, without loss of generality, we consider a basic construct b to be L WHEN ξ, where L is the set of all object constructs in b. We assume $\xi = \top$ if the WHEN sub-clause is not given in the query. It should be mentioned that the WHEN sub-clause is not syntactic sugar. Since basic queries form the scope for variables, the WHEN sub-clause can not be rewritten to a full query with UNION.

The semantics of a basic construct is defined as:

$$[\![L \text{ WHEN } \xi]\!]^{\text{graph}}_{\Omega,G} = \begin{cases} [\![L]\!]^{\text{graph}}_{\Omega,G} & \text{if } [\![\xi]\!]_{\Omega^\xi,G} = \top \\ G_\emptyset & \text{otherwise,} \end{cases}$$

$$[\![L \text{ WHEN } \xi]\!]^{\text{bindings}}_{\Omega,G} = \begin{cases} [\![L]\!]^{\text{bindings}}_{\Omega,G} & \text{if } [\![\xi]\!]_{\Omega^\xi,G} = \top \\ \{\mu_\emptyset\} & \text{otherwise.} \end{cases}$$

where $\Omega^\xi = \Omega \bowtie [\![L \text{ WHEN } \xi]\!]^{\text{bindings}}_{\Omega,G}$. Note, that Ω^ξ ensures that the expression ξ in a WHEN sub-clause can use variables appearing the MATCH clause (B) as well as construct variables of the constructs.

For the semantics of L, let L_N be the set of all node constructs in L and L_R the set of all relationship constructs (edge and path constructs) in L. We define the semantics of L such that the PPGs and the sets of bindings resulting from evaluation of all object constructs in L are united and joined, respectively. The union of graphs is defined as in Section A.5. Further, we ensure that the evaluation of all relationship constructs (L_R) is based on the set of bindings

resulting from the evaluation of all node constructs (L_N), so new edges connect to new nodes. The semantics of L is:

$$\begin{aligned}[\![L]\!]^{\text{graph}}_{\Omega,G} &= G_N \cup \left(\bigcup_{g\in L_R} [\![g]\!]^{\text{graph}}_{\Omega_N,G}\right)\\ [\![L]\!]^{\text{bindings}}_{\Omega,G} &= \Omega_N \bowtie \left(\bowtie_{g\in L_R} [\![g]\!]^{\text{bindings}}_{\Omega_N,G}\right)\end{aligned}$$

with:

$$\begin{aligned}G_N &= [\![L_N]\!]^{\text{graph}}_{\Omega,G} = \bigcup_{g\in L_N} [\![g]\!]^{\text{graph}}_{\Omega,G}\\ \Omega_N &= \Omega \bowtie [\![L_N]\!]^{\text{bindings}}_{\Omega,G} = \bowtie_{g\in L_N} [\![g]\!]^{\text{bindings}}_{\Omega,G}.\end{aligned}$$

In the following we define the semantics object constructs and with that $[\![g]\!]^{\text{graph}}_{\Omega,G}$ and $[\![g]\!]^{\text{bindings}}_{\Omega,G}$.

Semantics of Object Construct Patterns. Let Ω be a set of bindings on PPG $G = (N, E, P, \rho, \delta, \lambda, \sigma)$. Given an object construct g with a grouping set Γ, we use $\texttt{grp}(\Omega, g)$ to denote the grouping of Ω according to the grouping set Γ of g. The grouped set of bindings $\texttt{grp}(\Omega, g)$ is the set of equivalence classes of Ω according to the equivalence relation $\sim_\Gamma$ such that $\mu_1 \sim_\Gamma \mu_2$ if $\forall x \in \Gamma, \mu_1(x) = \mu_2(x)$ with $\mu_1, \mu_2 \in \Omega$.

In the following, we denote one equivalence class of $\texttt{grp}(\Gamma, g)$ as Ω' so that $\Omega' \in \texttt{grp}(\Gamma, g)$ holds. Ω' is a set of bindings, where all $\mu \in \Omega'$ bind a grouping variable $x \in \Gamma$ to the same value. We denote the value of a grouping variable x in Ω' as $\Omega'(x)$. Further, we denote the projection of Ω' to the grouping variables Γ as $\Omega'(\Gamma)$ such that $\Omega'(\Gamma) = \{x \mapsto \Omega'(x) \mid x \in \Gamma\}$.

The semantics of an object construct g with grouping set Γ with respect to Ω evaluated on G is the pair $\left([\![g]\!]^{\text{graph}}_{\Omega,G}, [\![g]\!]^{\text{bindings}}_{\Omega,G}\right)$ where $[\![g]\!]^{\text{graph}}_{\Omega,G}$ is the PPG defined as

$$[\![g]\!]^{\text{graph}}_{\Omega,G} = \bigcup_{\Omega'\in\texttt{grp}(\Omega,g)} [\![g]\!]^{\text{graph}}_{\Omega',G}$$

and $[\![g]\!]^{\text{bindings}}_{\Omega,G}$ is the set of bindings defined as

$$[\![g]\!]^{\text{bindings}}_{\Omega,G} = \bigcup_{\Omega'\in\texttt{grp}(\Omega,g)} [\![g]\!]^{\text{bindings}}_{\Omega',G}.$$

We next define the semantics of $[\![g]\!]^{\text{graph}}_{\Omega',G}$ and $[\![g]\!]^{\text{bindings}}_{\Omega',G}$ for each object construct, assuming $G = (N, E, P, \rho, \delta, \lambda, \sigma)$.

- If g is a node construct (x Group $\Gamma; S$), then

$$\begin{aligned}[\![g]\!]^{\text{graph}}_{\Omega',G} &= (\{v\}, \emptyset, \emptyset, \emptyset, \emptyset, [\![S]\!]^{\text{labels}}_{x,v,\Omega',G}, [\![S]\!]^{\text{properties}}_{x,v,\Omega',G})\\ [\![g]\!]^{\text{bindings}}_{\Omega',G} &= \{\{x \mapsto v\} \cup \Omega'(\Gamma)\}\end{aligned}$$

  where $v = \begin{cases}\Omega'(x) & \text{if } x \in B \text{ and } \Omega'(x) \text{ is defined}\\ \texttt{new}(x, \Omega'(\Gamma)) & \text{if } x \notin B.\end{cases}$

  Here $\texttt{new}(x, \Omega'(\Gamma))$ is a skolem function, returning a new distinct identifier for distinct values of x and $\Omega'(\Gamma)$. In case $x \in B$ and $\Omega'(x)$ is undefined, then $[\![g]\!]^{\text{graph}}_{\Omega',G} = G_\emptyset$ and $[\![g]\!]^{\text{bindings}}_{\Omega',G} = \{\mu_\emptyset\}$.
- If g is an edge construct $x \xrightarrow{z \text{ Group } \Gamma;S} y$ and $\Omega'(x)$ and $\Omega'(y)$ are defined, then

$$\begin{aligned}[\![g]\!]^{\text{graph}}_{\Omega',G} &= (\{v, u\}, \{e\}, \emptyset, \{e \mapsto (v, u)\}, \emptyset, [\![S]\!]^{\text{labels}}_{z,e,\Omega',G}, [\![S]\!]^{\text{properties}}_{z,e,\Omega',G})\\ [\![g]\!]^{\text{bindings}}_{\Omega',G} &= \{\{z \to e\} \cup \Omega'(\Gamma)\}\end{aligned}$$

where $e = \begin{cases} \Omega'(z) & \text{if } z \in B \text{ and } \Omega'(z) \text{ is defined} \\ \mathsf{new}(z, \Omega'(\Gamma)) & \text{if } z \notin B \end{cases}$

and $v = \Omega'(x)$ and $u = \Omega'(y)$. In case $\Omega'(x)$ or $\Omega'(y)$ is undefined, or in case $x \in B$ and $\Omega'(x)$ is undefined, then $[\![g]\!]^{\text{graph}}_{\Omega',G} = G_\emptyset$ and $[\![g]\!]^{\text{bindings}}_{\Omega',G} = \{\mu_\emptyset\}$. This prevents dangling edges.

- If g is a path construct $x \xrightarrow{@u;S} y$ and $\Omega'(u)$ is defined, then

$$[\![g]\!]^{\text{graph}}_{\Omega',G} = (N', E', \{p\}, \{e \mapsto \rho(e) \mid e \in \mathsf{edges}(p)\}, \{p \mapsto \delta(p)\}, \lambda', \sigma')$$
$$[\![g]\!]^{\text{bindings}}_{\Omega',G} = \{\Omega'(\{x, y\}) \cup \{u \mapsto p\}\}$$

  where $p = \Omega'(u)$ and

$$\lambda' = \lambda|_{\mathsf{nodes}(u) \cup \mathsf{edges}(u)} \cup [\![S]\!]^{\text{labels}}_{u,p,\Omega',G}$$
$$\sigma' = \sigma|_{\mathsf{nodes}(u) \cup \mathsf{edges}(u)} \cup [\![S]\!]^{\text{properties}}_{u,p,\Omega',G}$$

  If $\Omega'(u)$ is undefined, then $[\![g]\!]^{\text{graph}}_{\Omega',G} = G_\emptyset$ and $[\![g]\!]^{\text{bindings}}_{\Omega',G} = \{\mu_\emptyset\}$. Note that $\Omega'(x)$ and $\Omega'(y)$ are defined, if $\Omega'(u)$ is defined, since u, x, and y are all bound by the same path pattern in the MATCH clause.

- If g is a path construct $x \xrightarrow{u;S} y$ and $\Omega'(u)$ is defined, then

$$[\![g]\!]^{\text{graph}}_{\Omega',G} = (\mathsf{nodes}(u), \mathsf{edges}(u), \emptyset, \{e \mapsto \rho(e) \mid e \in \mathsf{edges}(p)\}, \emptyset, \lambda', \sigma')$$
$$[\![g]\!]^{\text{bindings}}_{\Omega',G} = \{\Omega'(\{x, y\})\}$$

  where $\lambda' = \lambda|_{\mathsf{nodes}(u) \cup \mathsf{edges}(u)}$ and $\sigma' = \sigma|_{\mathsf{nodes}(u) \cup \mathsf{edges}(u)}$. If $\Omega'(u)$ is undefined, then $[\![g]\!]^{\text{graph}}_{\Omega',G} = G_\emptyset$ and $[\![g]\!]^{\text{bindings}}_{\Omega',G} = \{\mu_\emptyset\}$.

Set and Remove Assignments. In the practical syntax, SET and REMOVE assignments can appear either inlined in the object pattern or in the SET–REMOVE sub-clause of the CONSTRUCT clause. In both cases, an assignment refers to a particular object construct by means of a construct variable. In the following, we assume all assignments are inlined and appear in set of assignments S of the respective object construct.

For a construct variable x, a variable $y \in B \cap (\mathcal{N} \cup \mathcal{E} \cup \mathcal{P})$, a label $l \in \mathbf{L}$, a property key $k \in \mathbf{K}$, and an expression ξ, assignments are syntactically defined as follows:

- $(+x = y)$, $(+x : l)$, and $(+x.k = \xi)$ are SET assignments;
- $(-x : l)$, and $(-x.k)$ are REMOVE assignments.

For a variable x and an object identifier o, the semantics of S is defined as $[\![S]\!]^{\text{labels}}_{x,o,\Omega',G} = (\lambda\ |_o \cup \lambda_S) \setminus \lambda_R$ and $[\![S]\!]^{\text{properties}}_{x,o,\Omega',G} = (\sigma\ |_o \cup \sigma_S) \setminus \sigma_R$ where $\lambda\ |_o$ is defined as expected and $\sigma\ |_o =$

$\{(o,k) \mapsto v \mid (o,k) \mapsto v \in \sigma \wedge \neg\exists v', (o,k) \mapsto v' \in \sigma_S\}$, and

$$\begin{aligned}
\lambda_S &= \big\{o \mapsto l \mid \Omega'(y) \mapsto l \in \lambda \wedge (+x = y) \in S\big\} \\
&\quad \cup \{o \mapsto l \mid (+x : l) \in S\} \\
\lambda_R &= \{o \mapsto l \mid (-x : l) \in S\} \\
\sigma_S &= \big\{(o,k) \mapsto v \mid \big(\Omega'(y), k\big) \mapsto v \in \sigma \wedge (+x = y) \in S\big\} \\
&\quad \cup \big\{(o,k) \mapsto [\![\xi]\!]_{\Omega',G} \mid (+x.k = \xi) \in S\big\} \\
\sigma_R &= \{(o,k) \mapsto v \mid (o,k) \mapsto v \in \sigma \wedge (-x.k) \in S\}\,.
\end{aligned}$$

In essence, all SET assignments override values of existing properties k and are applied before all REMOVE assignments, so that any expression ξ used in a SET assignment sees all existing properties.

Full Construction Patterns. We now have the necessary ingredients to define the semantics of CONSTRUCT clauses. The syntax of such expressions is given by the following grammar:

$$\begin{aligned}
\textit{constructClause} &::= \textsc{Construct}\ \textit{fullConstruct} \\
\textit{fullConstruct} &::= \textit{basicConstruct} \mid \textit{basicConstruct}, \textit{fullConstruct}
\end{aligned}$$

Note that we are not considering SET–REMOVE sub-clauses here.

Given a full construct f consisting of a set of basic constructs L, we define the semantics of CONSTRUCT clause as follow.

$$[\![\textsc{Construct}\ f]\!]_{\Omega,G} = \bigcup_{b \in L} [\![b]\!]^{\text{graph}}_{\Omega,G}$$

Example. Consider a CONSTRUCT $\{f, g, h\}$ WHEN $\top$ as shown in Line 21 without `social_graph`, where

$$f = (x\ \textsc{Group}\ \{e\}\,;\{(+x : \text{Company})\,, (+x.\text{name} = e)\}),$$
$$g = (n\ \textsc{Group}\ \{n\}\,;\emptyset),\ \text{and}\ h = n \xrightarrow{y\ \textsc{Group}\ \{x,e,n\};\{(+e:\text{worksAt})\}} x.$$

Let's assume the MATCH clause has the following set of bindings when evaluated on the PPG G in Figure 2:

$$\begin{aligned}
\Omega = \{\ &\{(n \mapsto \#\textbf{Frank}), (e \mapsto \#\textbf{MIT})\}\,, \{(n \mapsto \#\textbf{Frank}), (e \mapsto \#\textbf{CWI})\}\,, \\
&\{(n \mapsto \#\textbf{Alice}), (e \mapsto \#\textbf{Acme})\}\,, \{(n \mapsto \#\textbf{Celine}), (e \mapsto \#\textbf{HAL})\}\,, \{(n \mapsto \#\textbf{John}), (e \mapsto \#\textbf{Acme})\}\ \}.
\end{aligned}$$

We then have $[\![\{f, g, h\}]\!]_{\Omega,G} = G_N \cup [\![h]\!]^{\text{graph}}_{\Omega_N,G}$ with

$$
\begin{aligned}
G_N &= [\![f]\!]^{\text{graph}}_{\Omega,G} \cup [\![g]\!]^{\text{graph}}_{\Omega,G} \\
&= (\{\text{\#HAL, \#Acme, \#MIT, \#CWI}\}, \emptyset, \emptyset, \emptyset, \emptyset, \quad \{\text{\#HAL} \mapsto \text{:Company}, \ldots\}, \{(\text{\#HAL, name}) \mapsto \text{"HAL"}, \ldots\}) \\
&\cup (\{\text{\#John, \#Frank, \#Alice, \#Celine}\}, \emptyset, \emptyset, \emptyset, \emptyset, \quad \{\text{\#John} \mapsto \text{:Person}, \ldots\}, \{(\text{\#John, fristname}) \mapsto \text{"John"}, \ldots\}) \\
&= (\{\text{\#HAL, \#Acme, \#MIT, \#CWI, \#John, \#Frank}, \ldots\}, \ldots) \\
\Omega_N &= [\![(x \textsc{ Group } \{e\}; S_x)]\!]^{\text{bindings}}_{\Omega,G} \bowtie [\![(n \textsc{ Group } \{n\}; \emptyset)]\!]^{\text{bindings}}_{\Omega,G}
\end{aligned}
$$

$$
= \Omega \bowtie \begin{array}{|l|l|}\hline \textbf{x} & \textbf{e} \\ \hline \text{\#HAL} & \text{"HAL"} \\ \text{\#Acme} & \text{"Acme"} \\ \text{\#MIT} & \text{"MIT"} \\ \text{\#CWI} & \text{"CWI"} \\ \hline \end{array} \bowtie \begin{array}{|l|}\hline \textbf{n} \\ \hline \text{\#Frank} \\ \text{\#Alice} \\ \text{\#Celine} \\ \text{\#John} \\ \hline \end{array} = \begin{array}{|l|l|l|}\hline \textbf{n} & \textbf{x} & \textbf{e} \\ \hline \text{\#Frank} & \text{\#MIT} & \text{"MIT"} \\ \text{\#Frank} & \text{\#CWI} & \text{"CWI"} \\ \text{\#Alice} & \text{\#Acme} & \text{"Acme"} \\ \text{\#Celine} & \text{\#HAL} & \text{"HAL"} \\ \text{\#John} & \text{\#Acme} & \text{"Acme"} \\ \hline \end{array}
$$

so that $[\![h]\!]_{\Omega_N,G} = (N, E, \emptyset, \rho, \emptyset, \lambda, \emptyset)$ with

$$
\begin{aligned}
N &= \{\text{\#John, \#Acme, \#Frank, \#MIT, \#Frank, \#CWI}, \ldots\}, \\
E &= \{\text{\#1, \#2, \#3, \#4, \#5}\}, \\
\rho &= \{\text{\#1} \mapsto (\text{\#John, \#Acme}), \text{\#2} \mapsto (\text{\#Frank, \#MIT}), \text{\#2} \mapsto (\text{\#Frank, \#CWI}), \ldots\}, \text{ and} \\
\lambda &= \{\text{\#1} \mapsto \text{:worksAt}, \ldots\}.
\end{aligned}
$$

Finally, $[\![\{f, g, h\}]\!]_{\Omega,G} = (N, E, \emptyset, \rho, \emptyset, \lambda, \sigma)$ with

$$
\begin{aligned}
\lambda &= \{\text{\#John} \mapsto \text{:Person, \#Acme} \mapsto \text{:Company, \#1} \mapsto \text{:worksAt}, \ldots\}, \text{ and} \\
\sigma &= \{(\text{\#John, fristname}) \mapsto \text{"John"}, (\text{\#Acme, name}) \mapsto \text{"Acme"}, \ldots\}.
\end{aligned}
$$

A.4 The PATH clause

A frequent idiom is to define a temporary view consisting of paths specified using a *path pattern*, optionally associating a weight to each path in the view. The grammar for path clause expressions is:

$$
\begin{aligned}
\textit{basicPathClause} &::= \textsc{Path } \textit{pname} = \textit{pathPattern} \\
\textit{basicPathClauseCost} &::= \textit{basicPathClause } \textsc{Cost } \textit{fexpr} \\
\textit{pathPattern} &::= \textit{walkPattern} \mid \textit{walkPattern}; \textit{ graphPattern} \\
\textit{graphPattern} &::= \textit{basicGraphPattern} \mid \textit{basicGraphPattern}, \textit{ graphPattern} \\
\textit{walkPattern} &::= \textit{edgePattern} \mid \textit{edgePattern}, \textit{ walkPattern} \mid \\
&\qquad \textit{pathPattern} \mid \textit{pathPattern}, \textit{ walkPattern}
\end{aligned}
$$

where *pname* is a path view name and *fexpr* is a cost function expression. Note that *pathPattern* can refer to path views defined by other Path clauses appearing before it in the head clause where it is defined.

In the syntax, we additionally require that walk patterns are joinable. Given a walk pattern $b_1, \ldots, b_n$, where $n \geq 1$ and each b_i is either an edge pattern or a path pattern, we say that $b_1, \ldots, b_n$ is joinable if for every $i \in \{1, \ldots, n-1\}$:

- b_i is either an edge pattern $x \xrightarrow{z} y$ or a path pattern $x \xRightarrow{@w \text{ in } r} y$ or a path pattern $x \xRightarrow{w \text{ in } r} y$, b_{i+1} is either an edge pattern $x' \xrightarrow{z'} y'$ or a path pattern $x' \xRightarrow{@w' \text{ in } r'} y'$ or a path pattern $x' \xRightarrow{w' \text{ in } r'} y'$, and $y = x'$.

The interpretation of a walk pattern in a graph G is a path in G.
We express path views via a PATH clause, defined as follows:

$$\begin{aligned} \textit{pathClause} ::= {} & \textit{basicPathClause} \mid \textit{basicPathClauseCost} \mid \\ & \textit{basicPathClause} \text{ WHERE } \textit{BooleanCondition} \mid \\ & \textit{basicPathClauseCost} \text{ WHERE } \textit{BooleanCondition} \end{aligned}$$

Intuitively, the evaluation of

$$\text{PATH } \textit{pname} = \textit{pathPattern} \text{ COST } \textit{fexpr} \text{ WHERE } \textit{BooleanCondition}$$

on a PPG G is a view *pname* consisting of a set of paths, each satisfying *BooleanCondition*. The paths are given by bindings of *pathPattern* = *walkPattern*; *graphPattern* in G. More precisely, assuming that $x, y \in \mathcal{N}$ are the starting and ending nodes of *walkPattern*, for every pair of nodes a, b in G, we look for the shortest path L from a to b over G that conforms to *walkPattern* and satisfies the other graph patterns in *pathPattern*, and if such a path exists then we check that it satisfies *BooleanCondition*. If this holds, then we create a binding μ and a fresh path identifier p such that $\text{dom}(\mu) = \{x, y, \textit{pname}\}$, $\mu(x) = a$, $\mu(y) = b$, $\mu(\textit{pname}) = p$ and p is associated to L. Finally, the evaluation of the path clause is a set containing all such bindings μ. It is straightforward to see how these ideas can be formalized by using the notation considered previously.

Regarding the use of path views, recall that *pname* is defined in the head clause of a full graph query φ. For any path pattern appearing in φ, *pname* can be used as a symbol in the regular expression over which the pattern is evaluated. *pname* is then prefixed with the tilde symbol "~" to indicate that reference is being made to the binary view $\textit{pname}(x, y)$ (rather than an element of **L**).

A.5 Basic and Full Graph Queries

Recall that a basic graph query is defined as a CONSTRUCT clause followed by a MATCH clause, i.e., a basic graph query has the form CONSTRUCT Φ MATCH Ψ. The evaluation of such a query over a graph G is defined as: $[\![\text{CONSTRUCT } \Phi \text{ MATCH } \Psi]\!]_G = [\![\Phi]\!]_{\Omega, G}$, where $\Omega = [\![\Psi]\!]_{\Omega', G}$ and Ω' is either the set of bindings provided by an outer query if Ψ is an inner query or a singleton set containing the binding $\mu_\emptyset$ with empty domain (recall that $\mu_\emptyset$ is compatible with every other binding). A full graph query is obtained by using the operators UNION, INTERSECT, and MINUS to combine basic graph queries. Thus, to complete the definition of full graph queries, we need to provide the semantics of these operators on PPG's.

In what follows, assume that $G_1 = (N_1, E_1, P_1, \rho_1, \delta_1, \lambda_1, \sigma_1)$ and $G_2 = (N_2, E_2, P_2, \rho_2, \delta_2, \lambda_2, \sigma_2)$ are PPG. Moreover, say that G_1 and G_2 are consistent if: (i) for every $e \in E_1 \cap E_2$, it holds that $\rho_1(e) = \rho_2(e)$, and (ii) for every $p \in P_1 \cap P_2$, it holds that $\delta_1(p) = \delta_2(p)$.

The union operator. If G_1 and G_2 are not consistent, then $G_1 \cup G_2$ is defined as the empty PPG. Otherwise, $G_1 \cup G_2 = (N_1 \cup N_2, E_1 \cup E_2, P_1 \cup P_2, \rho, \delta, \lambda, \sigma)$, where (i) for every $x \in (N_1 \cup N_2 \cup E_1 \cup E_2 \cup P_1 \cup P_2)$ and $k \in \mathbf{K}$: $\lambda(x) = \lambda_1(x) \cup \lambda_2(x)$

and $\sigma(x,k) = \sigma_1(x,k) \cup \sigma_2(x,k)$; (ii) for every $e \in E_1 \cup E_2$:

$$\rho(e) \quad = \quad \begin{cases} \rho_1(e) & \text{if } e \in E_1 \\ \rho_2(e) & \text{otherwise;} \end{cases}$$

and (iii) for every $p \in P_1 \cup P_2$:

$$\delta(p) \quad = \quad \begin{cases} \delta_1(p) & \text{if } p \in P_1 \\ \delta_2(p) & \text{otherwise.} \end{cases}$$

The intersection operator. If G_1 and G_2 are not consistent, then $G_1 \cap G_2$ is defined as the empty PPG. Otherwise, $G_1 \cap G_2 = (N_1 \cap N_2, E_1 \cap E_2, P_1 \cap P_2, \rho, \delta, \lambda, \sigma)$, where (i) for every $x \in (N_1 \cap N_2) \cup (E_1 \cap E_2) \cup (P_1 \cap P_2)$ and $k \in \mathbf{K}$: $\lambda(x) = \lambda_1(x) \cap \lambda_2(x)$ and $\sigma(x,k) = \sigma_1(x,k) \cap \sigma_2(x,k)$; (ii) for every $e \in E_1 \cap E_2$: $\rho(e) = \rho_1(e)$; and (iii) for every $p \in P_1 \cap P_2$: $\delta(p) = \delta_1(p)$.

The difference operator. The difference of G_1 and G_2 is defined as: $G_1 \setminus G_2 = (N, E, P, \rho, \delta, \lambda, \sigma)$, where: $N = N_1 \setminus N_2$, $E = \{e \in E_1 \setminus E_2 \mid \rho_1(e) = (a,b), a \in N \text{ and } b \in N\}$, $P = \{p \in P_1 \setminus P_2 \mid \mathsf{nodes}(p) \subseteq N \text{ and } \mathsf{edges}(p) \subseteq E\}$, $\rho = \rho_1|_E$, $\delta = \delta_1|_P$, $\lambda = \lambda_1|_{(N \cup E \cup P)}$ and $\sigma = \sigma_1|_{(N \cup E \cup P) \times \mathbf{K}}$.

A.6 The GRAPH clause and GRAPH VIEW

The Graph View clause is used to create permanent views that may be re-used by multiple queries while the GRAPH clause is used for the declaration of query-local views only (similar to the Path clause). Both clauses are given by the following grammar:

$$\begin{aligned} \textit{graphClause} \quad &::= \quad \textsc{Graph}\ \textit{gid}\ \textsc{As}\ (\textit{fullGraphQuery}), \\ \textit{graphView} \quad &::= \quad \textsc{Graph View}\ \textit{gid}\ \textsc{As}\ (\textit{fullGraphQuery}), \end{aligned}$$

Assuming that Ψ is a full graph query, the evaluation of clause Graph *gid* As (Ψ) over a graph G associates the graph $[\![\Psi]\!]_G$ to the graph identifier *gid*, that is, $\text{gr}(\textit{gid}) = [\![\Psi]\!]_G$. The semantics of Graph View *gid* As (Ψ) is defined exactly in the same way.