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
Pontificia Universidad Católica de Chile

Pablo Barceló
DCC, Universidad de Chile

Peter Boncz
CWI, Amsterdam

George Fletcher
Technische Universiteit Eindhoven

Claudio Gutierrez
DCC, Universidad de Chile

Tobias Lindaaker
Neo4j

Marcus Paradies†
DLR

Stefan Plantikow
Neo4j

Juan Sequeda
Capsenta

Oskar van Rest
Oracle

Hannes Voigt
Technische Universität Dresden

ABSTRACT

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.

CCS CONCEPTS

• **Information systems** → **Graph-based database models**; **Query languages for non-relational engines**; • **Theory of computation** → *Database query languages (principles)*;

KEYWORDS

Graph databases; graph data models; graph query languages

ACM Reference Format:
Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE A Core for Future Graph Query Languages: Designed by the LDBC Graph Query Language Task Force. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3183713.3190654

*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.
†Work performed while at SAP SE.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190654

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.

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 [4], Amazon Neptune [5], ArangoDB [11], Blazegraph [14], CosmosDB [26], DataStax Enterprise Graph [16], HANA Graph [38], JanusGraph [23], Neo4j [27], Oracle PGX [39], OrientDB [31], Sparksee [40], Stardog [41], TigerGraph [42], Titan [43], 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

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

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

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 [9] in order to establish standard benchmarks for such new graph data management systems. LDBC has since developed a number of graph data management benchmarks [18, 22, 24] 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 reuse and decomposition of queries. Current query languages do not provide full composability because they output tables of values, nodes or edges.

Paths 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 [17].

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. Because users regard path-finding an important feature, paths are outputs of certain G-CORE queries. The fact that G-CORE 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. Not only is this extension necessary to support both path-finding and compositionality, but it enables G-CORE to express queries on paths (e.g. match, filter on, join on and group on paths), which significantly extends its expressive power.

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 in *data complexity* [45], i.e., 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 the existence of intractable queries (which is a minimum requirement for any practical query language).

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 [18], which demonstrate its main features.

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

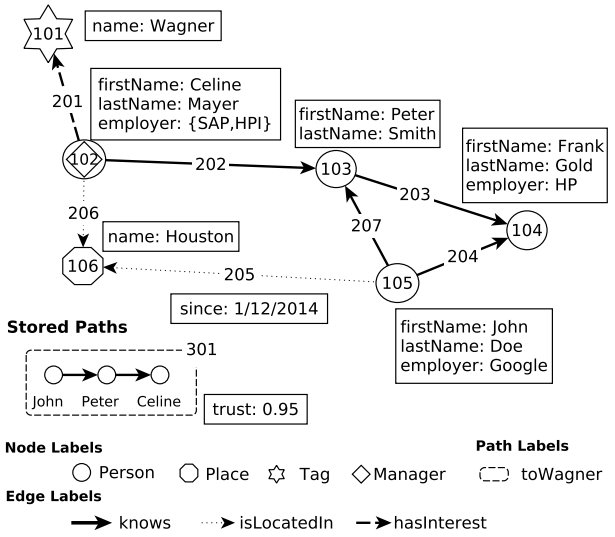

Figure 2: A small social network. A Path Property Graph (PPG) is a Property Graph that can have "Stored Paths".

We summarize our formal contributions, comprising syntax, semantics and complexity analysis of G-CORE in Section 4; further details on these are found in the technical report [7]. In Section 5, we show how G-CORE is extended to handle tabular data. We discuss related work in Section 6, and conclude 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 [8, 27, 28, 36, 46]. We call this model the *Path Property Graph* model, or PPG model for short. Let $\mathbf{L}$ be an infinite set of label names for nodes, edges and paths, $\mathbf{K}$ an infinite set of property names, and $\mathbf{V}$ an infinite set of literals (actual values such as integer and real numbers, strings, dates, truth values $\bot$ and $\top$, that represent true and false, respectively, etc.). Moreover, assume that `null` is a reserved word not occurring in $\mathbf{L} \cup \mathbf{K} \cup \mathbf{V}$. Finally, given a set X, let FSET(X) denote the set of all finite subsets of X, and FLIST(X) denote the set of all finite lists of elements from X.

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) $\sigma : (N \cup E \cup P) \times \mathbf{K} \to \text{FSET}(\mathbf{V}) \cup \{\texttt{null}\}$ is a total function such that: (i) $\sigma(x, k)$ is either a nonempty subset of $\mathbf{V}$ or the reserved word `null`, for every $(x, k) \in (N \cup E \cup P) \times \mathbf{K}$; and (ii) there exists a finite set of tuples $(y, \ell) \in (N \cup E \cup P) \times \mathbf{K}$ such that $\sigma(y, \ell) \neq \texttt{null}$.

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 $x \in (N \cup E \cup P)$, $k \in \mathbf{K}$ is a property name and $\sigma(x, k)$ is a nonempty subset of $\mathbf{V}$, then $\sigma(x, k)$ is the set of values of the property k for the identifier x. Otherwise, $\sigma(x, k) = \texttt{null}$, and property k is not defined for identifier x. Notice that although $\mathbf{K}$ is an infinite set of property names, in G only a finite number of properties are assigned actual values as we assume that there exists a finite set of tuples $(y, \ell) \in (N \cup E \cup P) \times \mathbf{K}$ such that $\sigma(y, \ell) \neq \texttt{null}$.

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. Notice that $\sigma(205, \text{name}) = \texttt{null}$ and $\sigma(301, \text{since}) = \texttt{null}$, as property name is not defined for the edge with identifier 205, while property since is not defined for the path with identifier 301.

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 nodes(p) and 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 nodes(p) = $[a_1, \ldots, a_{n+1}]$ and edges(p) = $[e_1, \ldots, e_n]$. In our example above, nodes(301) = [105, 103, 102] and edges(301) = [207, 202].

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.

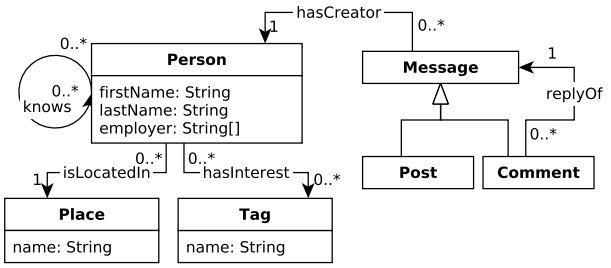

Figure 3: Social Network Benchmark schema (simplified).

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 [8].

Systems may omit `ON` if there is a *default* graph – let us assume in the sequel that `social_graph` is the default graph.[2] 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 [19] and the regular path expression syntax from PGQL [44], which has proven intuitive, effective and popular among property graph database users.

The previous example contains a `WHERE` filter whose semantics is defined by considering the fact that a property of a node, edge or path can have several values. More precisely, given that `n.employer` can be a set of values, the semantics of the condition `n.employer = 'Acme'` is defined by first interpreting `'Acme'` as the singleton set `{'Acme'}`, and then checking whether `n.employer` and `{'Acme'}` are the same set.[3] Thus, the condition `n.employer = 'Acme'` is used to check whether employee `n` works at Acme, and only in this company. Notice that if `n.employer` is equal to `null`, then the condition `n.employer = 'Acme'` does not hold. Moreover, if we want to retrieve the list of people who work at Acme, but who can also work at other companies, then we have to use the operator `IN` to check whether a value is an element of a set: `'Acme' IN n.employer`.

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

[2]By allowing a subquery enclosed in parentheses after the `ON` as an alternative to a graph identifier, G-CORE is a compositional language. With identical purpose it allows to use an identifier that denotes a `VIEW` – as will be demonstrated later.

[3]In general, we omit curly braces in the case of a singleton set, so we simply write `'Acme'` instead of `{'Acme'}`.

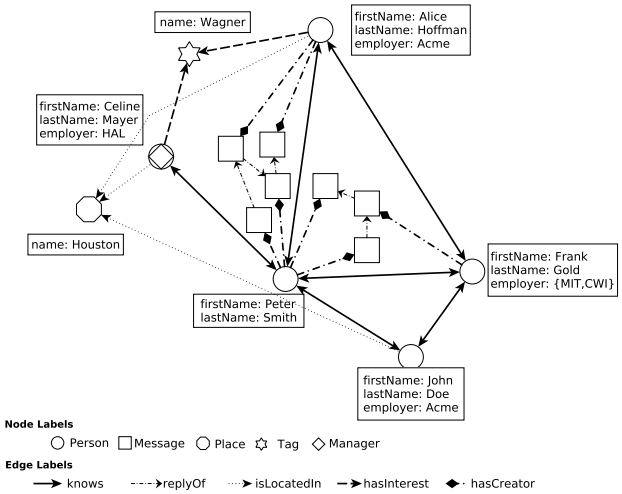

Figure 4: Initial graph database (`social_graph`). Note that the `knows` edges are drawn bi-directionally – this means there are two edges: one in each direction.

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 its 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, if `WHERE c.name = n.employer` were omitted, the query would be a Cartesian product, and would yield the below bindings for variables `c` and `n`, where we also indicate the values of expressions `c.name` and `n.employer`:

c.name	c	n	n.employer
"MIT"	#MIT	#Peter	null
"CWI"	#CWI	#Peter	null
"Acme"	#Acme	#Peter	null
"HAL"	#HAL	#Peter	null
"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 or `null`. But as mentioned before, 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`, as the sets `{"MIT"}` and `{"CWI"}` are different from the set `{"CWI","MIT"}`.

Notice that Peter is unemployed, so his `n.employer` value is `null`. More precisely, its Person node does not have an employer property at all. The absence of a property can be tested by using the Boolean built-in function `exists`; in our example, this condition would be `NOT exists(n.employer)`. 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 `e` to the `employer` property, which unrolls (or "explodes") 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, where `#Frank`'s two employers have been unrolled into individual bindings:

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

The "exploding" uses outerjoin semantics: absence of a property leads to a single `null` binding. Therefore a `null` binding of `e` is generated for `#Peter`, who has no employer. That binding is not in the above table because it does not qualify the `WHERE` condition `c.name = e` for any company: not only is there no company with `null` as name (i.e., without a name), but equi-comparisons with `null` yield `false` anyway, like in SQL.

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     WHEN exists(e)
23   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 (x:Company{name:=n.employer})<-[y:worksAt]-(n)
```

However, this unbound destination node `x` would create a company node for *each* binding[4]. 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.

Note that given the outerjoin semantics of property value binding (`employer=e`), we would now also create a company for the `null` binding of `e`. Therefore we used the `WHEN` clause that each `CONSTRUCT`

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

pattern can have, that restricts the bindings used for construction with a boolean predicate. This filters out the `null` bindings of `e`.

Our graph aggregation example query yields the same binding table for variables `n` and `e` as in the previous example. 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, `y` was unbound and could have been omitted. In the preceding examples, they were in fact omitted (we just used `[:worksat]`). Unbound variables in a `CONSTRUCT` are useful if they in *multiple* construct patterns, in order to ensure that the same, new, 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*>`:

```
24 CONSTRUCT (n)-/@p:localPeople{distance:=c}/->(m)
25   MATCH   (n)-/3 SHORTEST p<:knows*> COST c/->(m)
26     WHERE (n:Person) AND (m:Person)
27       AND n.firstName = 'John' AND n.lastName = 'Doe'
28       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.

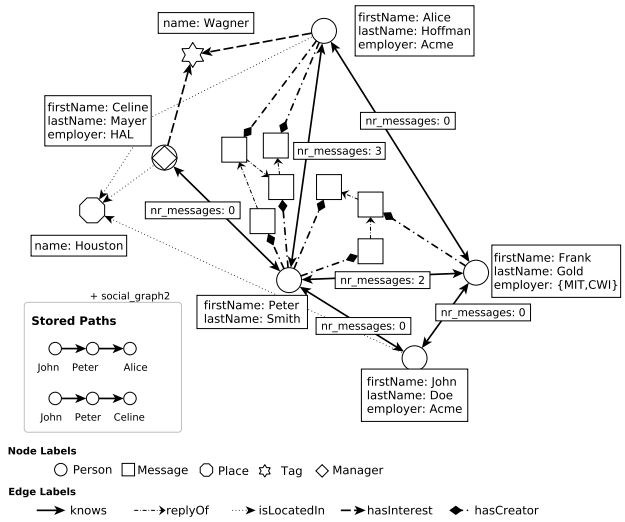

Figure 5: Graph view `social_graph1`, which adds `nr_message` properties to the original `social_graph` (`social_graph2` is `social_graph1` plus the Stored Paths in the grey box).

In `CONSTRUCT (n)-/@p:localPeople{distance:=c}/->(m)`, 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.

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:

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

In this case we use `-/<:knows*>/->` without the `SHORTEST` keyword. Using `ALL` instead of `SHORTEST`, which is asking for *all* paths, is not allowed if a path variable is bound to it and used somewhere, 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:

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

The method [12] 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:

```
37 WHERE EXISTS (
38   CONSTRUCT ()
39     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:

```
GRAPH VIEW social_graph1 AS (
  CONSTRUCT social_graph,
        (n)-[e]->(m) SET e.nr_messages := COUNT(*)
    MATCH (n)-[e:knows]->(m)
      WHERE (n:Person) AND (m:Person)
      OPTIONAL (n)<-[c1]-(msg1:Post|Comment),
               (msg1)-[:reply_of]-(msg2),
               (msg2:Post|Comment)-[c2]->(m)
        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. In G-CORE objects are mutable, i.e. not only newly constructed nodes, edges and paths can be modified that way but also ones bound by the `MATCH`, such as edge `e` in this example. In this example, the 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. 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:

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

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

Such a pattern is allowed in G-CORE, so the default way to evaluate a query containing multiple `OPTIONAL` blocks is from the top to the bottom. However, by imposing the simple and natural 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 [32]. Thus, a system implementing G-CORE can first check whether this condition is satisfied by a pattern with multiple `OPTIONAL` blocks, and if this is the case then it can use any order when evaluating these blocks; in particular, the system can decide which order to use based on estimations of the execution times of the different alternatives.

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`.

Matching	
Matching all patterns (Homomorphism)	*
Matching literal values	18, 23
Matching k shortest paths	25
Matching all shortest paths	30
Matching weighted shortest paths	61
(multi-segment) optional matching	45
Querying	
Querying multiple graphs	6
Queries on paths	70
Filtering matches	4,8,13,18,27,31,35,60,65,72
Filtering path expressions	59
Value joins	8
Cartesian product	11
List membership	13
Subqueries	
Set operations on graphs	8, 14, 19
Existential subqueries	
- Implicit	28, 32, 36
- Explicit	37
Construction	
Graph construction	*
Graph aggregation	21
Graph projection	24
Graph views	40, 58
Property addition	42

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

Powerful Path Patterns. Basic `PATH` patterns are a powerful building block that allow complex path expressions as concatenations of these patterns [44] 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, 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.[5] 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 (because of the use of `@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:

```
68 CONSTRUCT (n)-[e:wagnerFriend {score:=COUNT(*)}]->(m)
69            WHEN e.score > 0
70   MATCH    (n:Person)-/@p:toWagner/->(), (m:Person)
71     ON     social_graph2
72     WHERE m = nodes(p)[2]
```

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.

[5]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 `WHERE` condition cannot bind variables. The non-linear pattern also binds variable `c`, so it can be used for instance in a `COST` expression in G-CORE.

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:

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

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.

We provide the full detailed formal definitions of the syntax and semantics of G-CORE in the technical report [7]. 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* [35]).

Complexity analysis. The G-CORE query language has been carefully designed to ensure that G-CORE queries can be evaluated efficiently in data complexity [45]. 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. As extensively discussed in the database literature, cf. [3], this is a minimum requirement for any practical query language (as it ensures that no intractable properties about the data need to be evaluated).

The main reasons that explain the tractability in data complexity 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 a PPG whose label satisfies a fixed regular expression is an NP-complete problem [25]).

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 of 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 COMBINING GRAPHS AND TABLES

G-CORE is intentionally designed as a small language that provides a kernel of graph matching and construction functionality. Practical use of graphs, however, often requires interoperability with tabular data. For that purpose, we foresee integration of G-CORE with tabular querying. The formal semantics of G-CORE revolve around creation (by `MATCH`) and consumption (by `CONSTRUCT`) of its central concept, the binding *table*. This binding table also is the natural and clean interface point with queries on tabular data.

Tabular data output. G-CORE could support the `SELECT` clause, to be used in place of `CONSTRUCT`. The `SELECT` allows projecting expressions into a table. Precisely, `SELECT` is followed by one or many comma-separated expressions, whose main task is to facilitate turning node, edge and path variables into literal values – typically by accessing property values. The expressions follow the same syntax and semantics as the expressions allowed in G-CORE's `SET` clause.

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 single-column table with the names of these friends.

Tabular data input. To operate on tabular data, G-CORE could include a `FROM <table> <id>` clause similar to SQL, to be used in place of the `MATCH`. The binding table resulting from this would have one column by the name of the tuple variable `<id>` (or the table name if it were omitted). Its contents would be row identities from that table. The usable properties of these "row entities" would be all columns of `<table>`.

As an example consider the following query:

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

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

SQL+G-CORE. Taking this some steps further, we sketch how the above two extensions could be building blocks for extending SQL with G-CORE. Now, a SQL `FROM` clause may contain *multiple* table expressions, hence it introduces multiple tuple variables; this would lead to one column in the binding table for each such tuple variable, in terms of G-CORE's formal semantics of `CONSTRUCT`. Following SQL semantics, these multiple tables (or SQL views or SQL subqueries) are combined with Cartesian product, possibly filtered by `INNER/OUTER JOIN` predicates in the `FROM` clause. The `MATCH` clause could then also be employed *in conjunction* with the `FROM` clause (not only: *instead of*). The final binding table is the Cartesian product between the `MATCH` binding table and the `FROM` binding table, with the `WHERE` clause still available to restrict the combinations.

The following "SQL+G-CORE" query performs data integration between a graph and two tables, producing an enriched graph:

```
CONSTRUCT
  (c GROUP c.custId :Customer {custId:=c.custId}),
    (c)-[:bought {spent:=SUM(l.price)}]->(p)
    WHEN COUNT(l.productId) > 0
FROM customer c LEFT OUTER JOIN
     lineitem l USING (custId)
MATCH (p :Product) ON product_graph
WHERE p.productId = l.productId
```

Similarly, but not shown, the SQL `GROUP BY` and `HAVING` clauses can be used to aggregate and filter the binding table prior to `CONSTRUCT`.

Thus, this possible extension of SQL with G-CORE would add `MATCH` as an alternative way to create table expressions (each variable bound in `MATCH`, be it a node, edge or path variable, introducing a tuple variable), and `CONSTRUCT` as an alternative way to return a query result (as a graph).

This concludes our sketch of possible integration of tabular querying and G-CORE. The end result should be systems that can query both graphs and tables and return either a graph or a table; and thus achieve compositionality in both sorts of data.

At the time of this writing, the formal semantics of relational integration has been left out of scope in [7].

6 DISCUSSION AND RELATED WORK

Graph query languages have been extensively researched in the past decades, and comprehensive surveys are available. Angles and Gutierrez [10] surveyed GQLs proposed during the eighties and nineties, before the emergence of current (practical) graph database systems. Wood [47] studied GQLs focusing on their expressive power and computational complexity. Angles [6] compares graph database systems in terms of their support for essential graph queries. Barceló [13] studies the expressiveness and complexity of several navigational query languages. Recently, Angles et al. [8] 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 [20] is a graph-based programming language for property graphs which makes extensive use of XPath to support complex graph traversals. Cypher [15], 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 [19], while recognizing that Cypher is an evolving language where several advancements compared to Cypher 9 have already been made. Oracle has developed PGQL [44], 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 [39] and distributed systems [37], exist. Here, we consider PGQL 1.1 [30], which is the most recent version that is commercially available [29].

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, 33]. 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 [19].

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 [34], which was designed to support several types of complex graph patterns (including union and optional). Its latest version, SPARQL 1.1 [21], 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 [8]. 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 [4, 5, 11, 14, 16, 23, 26, 27, 31, 38–43].

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. Furthermore, we postulate that G-CORE can build upon existing relational technology, as a starting point, by using techniques such as views, recursions, etc. These efforts are currently underway by members of the LDBC Graph Query Language Task Force.

This paper is a call to action for the stakeholders driving the graph database industry.

ACKNOWLEDGEMENTS

The authors are grateful to the anonymous reviewers for their useful comments. R. Angles, M. Arenas, P. Barceló and C. Gutierrez were partially supported by the Millennium Nucleus Center for Semantic Web Research under grant NC120004, and the Millennium Institute for Foundational Research on Data. M. Arenas was also partially supported by the Fondecyt grant 1161473.

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] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
[4] AgensGraph - The Performance-Driven Graph Database. 2017. (2017). http://www.agensgraph.com/
[5] Amazon Neptune - Fast, reliable graph database build for cloud. 2017. (2017). https://aws.amazon.com/neptune/
[6] Renzo Angles. 2012. A comparison of current graph database models. In *4rd Int. Workshop on Graph Data Management: Techniques and Applications*.
[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2017. G-CORE: A Core for Future Graph Query Languages. *CoRR* abs/1712.01550 (2017). http://arxiv.org/abs/1712.01550
[8] 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
[9] 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.
[10] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.
[11] ArangoDB - Native multimodel database. 2017. (2017). https://arangodb.com/
[12] 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.
[13] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proc. of the 32nd Symposium on Principles of Database Systems (PODS)*. ACM, 175–188.
[14] Blazegraph. 2017. (2017). https://www.blazegraph.com/
[15] Cypher - Graph Query Language. 2017. (2017). http://neo4j.com/developer/cypher-query-language/
[16] DataStax Enterprise Graph. 2017. (2017). https://www.datastax.com/products/datastax-enterprise-graph
[17] 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.
[18] 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.
[19] 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. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2018)*.
[20] Gremlin - A graph traversal language. 2017. (2017). https://github.com/tinkerpop/gremlin
[21] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language - W3C Recommendation. https://www.w3.org/TR/sparql11-query/. (March 21 2013).
[22] 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.
[23] JanusGraph - Distributed graph database. 2017. (2017). http://janusgraph.org/
[24] 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
[25] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.
[26] Microsoft Azure Cosmos DB. 2017. (2017). https://docs.microsoft.com/en-us/azure/cosmos-db/introduction
[27] Neo4j. 2017. The Neo4j Developer Manual v3.3. (2017).
[28] The openCypher implementer's group. 2017. Property Graph Model. (2017). https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc
[29] Oracle. 2017. Oracle Big Data Spatial and Graph. (2017). http://www.oracle.com/technetwork/database/database-technologies/bigdata-spatialandgraph/
[30] Oracle. 2017. PGQL 1.1 Specification. (2017). http://pgql-lang.org/spec/1.1/
[31] OrientDB - Multi-Model Database. 2017. (2017). http://orientdb.com/
[32] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.
[33] 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
[34] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF - W3C Recommendation. https://www.w3.org/TR/rdf-sparql-query/. (2008).
[35] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2017. Regular Queries on Graph Databases. *Theory Comput. Syst.* 61, 1 (2017), 31–83.
[36] 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.
[37] 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).
[38] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database.. In *BTW*, Vol. 13. 403–420.
[39] 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.
[40] Sparksee - Scalable high-performance graph database. 2017. (2017). http://www.sparsity-technologies.com/#sparksee
[41] Stardog - The Knowledge Graph Platform for the Enterprise. 2017. (2017). http://www.stardog.com/
[42] TigerGraph - The First Native Parallel Graph. 2017. (2017). https://www.tigergraph.com/
[43] Titan - Distributed Graph Database. 2017. (2017). http://titan.thinkaurelius.com/
[44] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES2016*. ACM, 7.
[45] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *STOC*. 137–146.
[46] 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.
[47] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.