G-CORE
A Core for Future Graph Query Languages

Designed by the LDBC Graph Query Language Task Force†

© 2018 Association for Computing Machinery. To post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permissions permissions@acm.org.

Renzo Angles
Universidad de Talca

Marcelo Arenas
PUC Chile

Pablo Barceló
DCC, Universidad de Chile

Peter Boncz
CWI, Amsterdam

George Fletcher
Technische Universität 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

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 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 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 development.
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 represent 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 than 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 functionallity of the language, thus enabling the development of correct implementations. In particular, an open source grammar for G-CORE is available\(^1\).

Complextiy 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. 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 \(I\) 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

---

\(^1\)https://github.com/ldbc/log_parser
(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 \(\texttt{null}\) is a reserved word not occurring in \(L \cup K \cup V\). Finally, given a set \(X\), let \(\text{FSET}(X)\) denote the set of all finite subsets of \(X\), and \(\text{FLIST}(X)\) 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) \(\rho : E \rightarrow (N \times N)\) is a total function.

(3) \(\delta : P \rightarrow \text{FLIST}(N \cup E)\) is a total function such that for every \(p \in P\), it holds that \(\delta(p) = [a_1, e_1, a_2, \ldots, a_{n_p}, e_{n_p}, a_{n_p+1}]\), where:

- (i) \(n_p \geq 0\),
- (ii) \(e_j \in E\) for every \(j \in \{1, \ldots, n_p\}\),
- (iii) \(\rho(e_j) = (a_j, a_{j+1})\) or \(\rho(e_j) = (a_{j+1}, a_j)\) for every \(j \in \{1, \ldots, n_p\}\).

(4) \(\lambda : (N \cup E \cup P) \rightarrow \text{FSET}(L)\) is a total function.

(5) \(\sigma : (N \cup E \cup P) \times K \rightarrow \text{FSET}(V) \cup \{\text{null}\}\) is a total function such that:

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

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 \(\rho\) allows us to have several edges between the same pairs of nodes. Function \(\delta\) assigns to each path identifier \(p \in P\) an actual path in \(G\); this is a list \([a_1, e_1, a_2, \ldots, a_{n_p}, e_{n_p}, a_{n_p+1}]\) satisfying condition (3) in Definition 2.1. Function \(\lambda\) is used to specify the set of labels of each node, edge, and path, while function \(\sigma\) is used to specify the values of a property for every node, edge, and path. To be precise, if \(x \in (N \cup E \cup P), k \in K\) is a property name and \(\sigma(x, k)\) is a nonempty subset of \(V\), then \(\sigma(x, k)\) is the set of values of the property \(k\) for the identifier \(x\). Otherwise, \(\sigma(x, k) = \text{null}\), and property \(k\) is not defined for identifier \(x\). Notice that although \(K\) 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 \((y, \ell) \in (N \cup E \cup P) \times K\) such that \(\sigma(y, \ell) \neq \text{null}\).

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

\[
N = \{101, 102, 103, 104, 105, 106\}, \quad E = \{201, 202, 203, 204, 205, 206, 207\}, \quad P = \{301\}
\]

as node, edge, and path identifiers, respectively;

\[
\rho = \{201 \mapsto (102, 101), \ldots, 207 \mapsto (105, 103)\}
\]

\[
\delta = \{301 \mapsto [105, 207, 103, 202, 102]\}
\]

as edge and path assignments, respectively; and,

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

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 \(\sigma(205, \text{name}) = \text{null}\) and \(\sigma(301, \text{since}) = \text{null}\), 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 \(\text{nodes}(p)\) and \(\text{edges}(p)\) 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 \(\text{nodes}(p) = [a_1, \ldots, a_{n+1}]\) and \(\text{edges}(p) = [e_1, \ldots, e_n]\). In our example above, \(\text{nodes}(301) = [105, 103, 102]\) and \(\text{edges}(301) = [207, 202]\).

**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 LDLC 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 (: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. 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 AFA-art syntax from Cypher [19] and the regular path expression syntax from PQL [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. 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 (un- connected) 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)<-[c:worksAt]-(n)
6 MATCH (:Company) ON company_graph,
7 (:Person) ON social_graph

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.

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

![Graph diagram](image)

**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.

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:

<table>
<thead>
<tr>
<th>c</th>
<th>n</th>
<th>employer</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIT</td>
<td>#MIT</td>
<td>#Peter</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#Peter</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#Peter null</td>
</tr>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#Frank</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#Frank</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#CWI MIT</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#Frank</td>
</tr>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#Alice</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#Alice</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#Alice</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#Alice</td>
</tr>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#Acme</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#Acme</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#Acme</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#Acme</td>
</tr>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#John</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#John</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#John</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#John</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>c.name</th>
<th>c</th>
<th>n.employer</th>
</tr>
</thead>
<tbody>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#Peter</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#Peter null</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#Peter null</td>
</tr>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#Celine</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#Celine</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#Celine</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#Celine</td>
</tr>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#Celine</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#Celine</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#Celine</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#Celine</td>
</tr>
<tr>
<td>'MIT'</td>
<td>#MIT</td>
<td>#John</td>
</tr>
<tr>
<td>'CWI'</td>
<td>#CWI</td>
<td>#John</td>
</tr>
<tr>
<td>'Acme'</td>
<td>#CWI</td>
<td>#John</td>
</tr>
<tr>
<td>'HAL'</td>
<td>#HAL</td>
<td>#John</td>
</tr>
</tbody>
</table>

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, so 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"="{MIT,"MIT"}" and "MIT"="{MIT,"MIT"}" evaluate to FALSE, as the sets ("MIT") and ("MIT") are different from the set ("MIT","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 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:

<table>
<thead>
<tr>
<th>c</th>
<th>n</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMI</td>
<td>#Frank</td>
<td>&quot;MIT&quot;</td>
</tr>
<tr>
<td>#CWI</td>
<td>#Frank</td>
<td>&quot;CWI&quot;</td>
</tr>
<tr>
<td>#HAL</td>
<td>#Alice</td>
<td>&quot;Acme&quot;</td>
</tr>
<tr>
<td>#HAL</td>
<td>#Celine</td>
<td>&quot;HAL&quot;</td>
</tr>
<tr>
<td>Acme</td>
<td>#John</td>
<td>&quot;Acme&quot;</td>
</tr>
<tr>
<td>Acme</td>
<td>#John</td>
<td>&quot;Acme&quot;</td>
</tr>
</tbody>
</table>

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:Company (name='e'))<-[[:worksAt]-(y)
22 WHERE exists(x)
23 MATCH (n:Person (employer=x))

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='e'})<-[[:worksAt]-(y)

However, this unbound destination node x would create a company node for each binding4. 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=x), we would now also create a company for the null binding of e. Therefore we used the WHEN clause that each CONSTRUCT 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

4In addition, it would create a company with the name property with the values ("CWI","MIT").
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 `:-/3` syntax which makes a copy of the bound y edges (as well as the `:-/m` 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: Person ) -/3 <:knows*>/-> (m: Person )
25 MATCH (n: Person )-/3 <:knows*>/-> (m: Person )
26 WHERE n. firstName = 'John' AND n. lastName = 'Doe'
27 AND (n)-[:isLocatedIn]->()<[:isLocatedIn]-(m) 
28 WHERE (n): Person AND (m): Person
29 CONSTRUCT (n)-/3 <:knows*>/-> (m) 
```

In G-CORE, paths are demarcated with slashes `//-`. In the above example `<: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 `<: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 <`/->` 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 <`/->` could be omitted.

In `CONSTRUCT (n)-/3 <:knows*>/-> (m)`, we see the bound path variable `p`. The `3` 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 `WHERE` clause: `(n): Person` and every possible person `m`: Person 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`.

**Reachability and All Paths.** In a similar query where we just return `n`, 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): Person` and every possible person `m`: Person 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 Optionsals.** 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:

```
37 WHERE EXISTS (  
38 CONSTRUCT ( )  
39 MATCH (n): Person  
40 OPTIONAL (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`.

Figure 5: Graph view `social_graph1`, which adds `nr_message` properties to the original `social_graph` (social_graph2 is `social_graph` plus the Stored Paths in the grey box).
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 s 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 \((n):[e]\rightarrow(m)\) adds nothing new, but as
described before, performs implicit graph aggregation, where bind-
ings 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 pe-
ople 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 \(n:Person\) and \(has_creator\)). This query also demonstrates
the use of disjunctive label tests (m:Person). This query also demonstrates
the use of disjunctive label tests (m:Person).

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

\begin{verbatim}
49 MATCH (n:Person)
50 OPTIONAL (n)-[:worksAt]-(c)
51 OPTIONAL (n)-[:livesIn]-(a)
\end{verbatim}

we need to perform the following steps: evaluate \((n:Person)\) to gen-
erate 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:

\begin{verbatim}
52 MATCH (n:Person)
53 OPTIONAL (n)-[:livesIn]-(a)
54 OPTIONAL (n)-[:worksAt]-(c)
\end{verbatim}

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 \((n:Person)\):

\begin{verbatim}
55 MATCH (n:Person)
56 OPTIONAL (n)-[:worksAt]-(a)
57 OPTIONAL (n)-[:livesIn]-(a)
\end{verbatim}

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 re-
striction 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 evalu-
ation 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 Wag-
ner 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, be
cause 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
minimalized) is the sum of the costs of all path segments. If the COST
is omitted, it defaults to 1 (hop count).

\begin{verbatim}
58 GRAPH VIEW social_graph1 AS ( 
59 PATH wKnows = (a)-[:knows]->(y)
60 WHERE NOT 'Acme' IN y.employer
61 COST 1 / (1 + e.nr_messages)
62 CONSTRUCT social_graph1.1 =/9p:toWagner /->(m)
63 MATCH (n:Person)-/p<-wKnows+/>- (m:Person)
64 ON social_graph1
65 WHERE (m)-[:hasInterest]-->(:Tag { name = 'Wagner'})
66 AND (n)-[:isLocatedIn]->(l)<-[:isLocatedIn]->(m)
67 AND n.firstName = "John" AND n.lastName = 'Doe')
\end{verbatim}

The result of this graph view \(social_graph1\) 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 (query) 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 build-
ning 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.\(^2\) The result of this query is a view

\(^2\)Note that non-linear path patterns, such as PATH \((a)\rightarrow\{(b), (b)\rightarrow(c)\) add power
over linear patterns with existential filters: PATH \((a)\rightarrow\{(b) WHERE (b)\rightarrow(c)\),
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.
Wagner lovers have been materialized (because of the use of the features shown in the previous sections. Formally, a one
of the main goals of this paper is to provide a formal definition between John and Peter with score 2. Both via Peter, the result
of this query is a single path. For these direct friends we count how often they occur as the start of a direct friend of John
Doe. :toWagner

<table>
<thead>
<tr>
<th>Matching</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching all patterns (Homomorphism)</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching literal values</td>
<td>18, 23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching k shortest paths</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching all shortest paths</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching weighted shortest paths</td>
<td>61</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(multi-segment) optional matching</td>
<td>45</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Querying</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Querying multiple graphs</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queries on paths</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filtering matches</td>
<td>4, 8, 13, 18, 27, 31, 35, 60, 65, 72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filtering path expressions</td>
<td>59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value joins</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cartesian product</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List membership</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subqueries</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set operations on graphs</td>
<td>8, 14, 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Existential subqueries</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Implicit</td>
<td>28, 32, 36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Explicit</td>
<td>37</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Construction</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph construction</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph aggregation</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph projection</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph views</td>
<td>40, 58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property addition</td>
<td>42</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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/->(n)).

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:

$$\text{CONSTRUCT } (n) - [e: \text{wagnerFriend} \{ \text{score}: \text{COUNT}(*))]\rightarrow (m)$$

$$\text{WHERE } e. \text{score} > 0$$

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 $\Omega$, 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 $\varphi$ that defines the content of a Match clause over a PPG $G$ always corresponds to a set $\Omega$ of bindings, which is denoted by $[\varphi]_G$. The bindings in $\Omega$ can then be filtered by using boolean conditions specified in the Where clause.
- A Construct clause $\psi$ then takes as input both the PPG $G$ and the set of bindings $\Omega$, and produces a new PPG $H$, which is denoted by $[\psi]_{\Omega, G}$. Note that $G$ is also an input in the evaluation of $\psi$, as the set of bindings $\Omega$ 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 $\Omega$ of all bindings of a fixed graph pattern $\varphi$ 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 $\varphi$ by node and edge identifiers in $G$, respectively, and then for each path variable $\pi$ representing a path in $G$ from node $u$ to node $v$ whose label must conform to a regular expression $r$, we replace $\pi$ 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

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:

\[
\text{query ::= headClause fullGraphQuery} \\
\text{headClause ::= } e \mid \text{pathClause headClause} \\
\text{pathClause ::= } \text{graphClause headClause} \\
\text{graphClause ::= basicGraphQuery} \mid \\
\text{fullGraphQuery ::= basicGraphQuery } | \\
\text{setOp ::= Union | Intersect | Minus} \\
\text{basicGraphQuery ::= constructClause matchClause}
\]

$$\text{(fullGraphQuery setOp fullGraphQuery)}$$

$$\text{setOp ::= Union | Intersect | Minus}$$

$$\text{basicGraphQuery ::= constructClause matchClause}$$

The result of evaluating the graph pattern $\varphi$ that defines the content of a Match clause over a PPG $G$ always corresponds to a set $\Omega$ of bindings, which is denoted by $[\varphi]_G$. The bindings in $\Omega$ can then be filtered by using boolean conditions specified in the Where clause.

- A Construct clause $\psi$ then takes as input both the PPG $G$ and the set of bindings $\Omega$, and produces a new PPG $H$, which is denoted by $[\psi]_{\Omega, G}$. Note that $G$ is also an input in the evaluation of $\psi$, as the set of bindings $\Omega$ 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 $\Omega$ of all bindings of a fixed graph pattern $\varphi$ 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 $\varphi$ by node and edge identifiers in $G$, respectively, and then for each path variable $\pi$ representing a path in $G$ from node $u$ to node $v$ whose label must conform to a regular expression $r$, we replace $\pi$ 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 \( \varphi \) whose evaluation corresponds to a binary relation over the nodes of the input PPG \( G \).

By construction, the graph pattern \( \varphi \) 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 \( \varphi_i \)'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 \( \Omega \) 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 \( \Omega \) can be carried out in polynomial time by a straightforward extension of the aforementioned techniques for efficiently evaluating basic graph patterns. Finally, filtering \( \Omega \) 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 \( \text{EXISTS} \ Q \), 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 \( \Omega \) 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:

```
73 SELECT m.lastName = 'John' AND n.lastName = 'Doe'
74 WHERE m.firstName = 'John' AND n.lastName = 'Doe'
75 MATCH (n:Person)-[:knows]->(m:Person)
76 (m)-[:isLocatedIn]->(c:City)
```

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:

```
77 CONSTRUCT
78 ( cust GROUP o.custName : Customer { custId :=o.custId })
79 ( prod GROUP o.prodCode : Product { code :=o.prodCode })
80 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/EXCEPT 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:

```
82 CONSTRUCT
83 ( c GROUP c.custId : Customer { custId :=c.custId })
84 ( p )-[[:bought] { spent :=SUM(l.price) }] ->(p)
85 WHEN COUNT( l.productId ) = 0
86 FROM customer c LEFT OUTER JOIN
87 lineitem l USING ( custId )
88 MATCH ( p : Product ) ON product_graph
89 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 language 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: compostability, 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 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


