Formalizing Gremlin Pattern Matching Traversals in an Integrated Graph Algebra

Harsh Thakkar[1], Sören Auer[1,2], Maria-Esther Vidal[2]

[1] Smart Data Analytics Lab (SDA), University of Bonn, Germany
[2] TIB & Leibniz University of Hannover, Germany
{lastname}@cs.uni-bonn.de[1], dpunjani@di.uoa.gr[2]

Abstract. Graph data management (also called NoSQL) has revealed beneficial characteristics in terms of flexibility and scalability by differently balancing between query expressivity and schema flexibility. This peculiar advantage has resulted into an unforeseen race of developing new task-specific graph systems, query languages and data models, such as property graphs, key-value, wide column, resource description framework (RDF), etc. Present-day graph query languages are focused towards flexible graph pattern matching (aka sub-graph matching), whereas graph computing frameworks aim towards providing fast parallel (distributed) execution of instructions. The consequence of this rapid growth in the variety of graph-based data management systems has resulted in a lack of standardization. Gremlin, a graph traversal language, and machine provide a common platform for supporting any graph computing system (such as an OLTP graph database or OLAP graph processors). In this extended report, we present a formalization of graph pattern matching for Gremlin queries. We also study, discuss and consolidate various existing graph algebra operators into an integrated graph algebra.

Keywords: Graph Pattern Matching, Graph Traversal, Gremlin, Graph Algebra

1 Introduction

Upon observing the evolution of information technology, we can observe a trend from data models and knowledge representation techniques being tightly bound to the capabilities of the underlying hardware towards more intuitive and natural methods resembling human-style information processing. This evolution started with machine assembly languages, went over procedural programming, object-oriented methods and resulted in an ever more loosely coupling of data and code with relational data bases, declarative query languages and object-relational mapping (ORM). In recent years, we can observe an even further step in this evolution – graph

based data models, which organize information in conceptual networks. Graphs are valued distinctly, when it comes to choosing formalisms for modelling real-world scenarios such as biological, transport, communication and social networks due to their intuitive data model. In particular, the property graph data model is capable of representing complex domain networks [17].

Graph analysis tools have turned out to be one of pioneering applications in understanding these natural and man-made networks [7]. Graph analysis is carried out using graph processing techniques which ultimately boil down to efficient graph query processing. Graph Pattern Matching (GPM), also referred to as the sub-graph matching is the foundational problem of graph query processing. Many vendors have proposed a variety of (proprietary) graph query languages to demonstrate the solvability of graph pattern matching problem.

These modern graph query languages focus either on *traversal*, where traversers move over vertices and edges of a graph in a user defined fashion or on *pattern-matching*, where graph patterns are matched against the graph database. *Gremlin* [16] is one such modern graph query language, with a distinctive advantage over others that it offers both of these perspectives. This implies that a user can reap benefits of both declarative and imperative matching style within the same framework. Furthermore, conducting GPM in Gremlin can be of crucial importance in cases such as:

- Querying very large graphs, where a user is not completely aware of certain dataset-specific statistics of the graph (e.g., the number of `created` vs `knows` edges existing in the graph (ref. Figure 1));
- Creating optimal query plans, without the user having to dive deep into traversal optimization strategies.
- In application-specific settings such as a question answering [26], users express information needs (e.g., natural language questions) which can be better represented as graph pattern matching problems than path traversals.

In this work, we present an extended version of our earlier work [25] that contributes to establishing a formal base for a graph query algebra, by surveying and integrating existing graph query operators. The contributions of this work are in particular:

- We consolidate existing graph algebra operators from the literature and propose two new traversal operators into an integrated graph algebra.

– We formalize the graph pattern matching construct of the Gremlin query language.
– We provide a formal specification of pattern matching traversals for the Gremlin language, which can serve as a foundation for implementing a Gremlin based query compilation engine.

As a result, the formalization of graph query algebra supports the integration and interoperability of different graph data models [5] (e.g., executing SPARQL queries on top of Gremlin [23,24]), helps to prevent vendor lock in scenarios and boosts data management benchmarking efforts such as LITMUS [10,21,22] and the Linked Data Benchmark Council (LDBC) [2].

The remainder of this article is organised as follows: section 2 describes the preliminaries including property graphs, graph pattern matching, and foundations of relational algebra. section 3 elaborates on Gremlin as a traversal language and machine, and discusses synergy of GPM and its evaluation in Gremlin. section 4 proposes the Gremlin to graph algebra mapping algorithm. section 5 surveys related work. In section 6 we conclude this article and discuss future work.

2 Preliminaries

In this section, we present and summarise the mathematical concepts used in this article. Our notation closely follows [9] and extends [18] by adding the notion of vertex labels.

2.1 Property Graphs

Property graphs, also referred to as directed, edge-labeled, attributed multi-graphs, have been formally defined in a wide variety of texts, such as [1,8,11,17,15]. We adapt the definition of property graphs presented by [17]:

Definition 1 (Property Graph). *A property graph is defined as* $G = \{V, E, \lambda, \mu\}$*; where:*

– V *is the set of vertices,*
– E *is the set of directed edges such that* $E \subseteq (V \times Lab \times V)$ *where* Lab *is the set of Labels,*
– λ *is a function that assigns labels to the edges and vertices (i.e.* $\lambda : V \cup E \rightarrow \Sigma^*$*)*[3]*, and*

[3] set of strings (Σ^*)

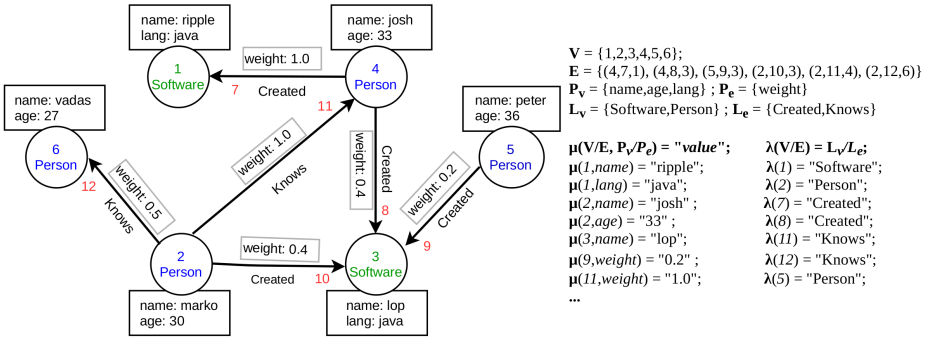

Fig. 1. Running example. This figure presents a collaboration network scenario of employees in a typical software company, There are 6 vertices (with labels "person" and "software") and 6 edges connecting them with two type of relations (i.e., edge labels) namely, "created" and "knows".

– *μ is a partial function that maps elements and keys to values (i.e. $\mu : (V \cup E) \times R \to S$) i.e. properties (key $r \in R$, value $s \in S$).* □

For simplicity, we define disparate sets (of μ and λ) for the labels and properties of vertices and edges respectively, adapting the terminology used in [9]. We define:

– L_v: Set of vertex labels ($L_v \subset \Sigma^*$), $\lambda_l : V \to L_v$ assigns label to each vertex
– L_e: Set of edge labels ($L_e \subset \Sigma^*$), $\lambda_e : E \to L_e$ assigns label to each edge. ($L_v \cap L_e = \phi$, $L_v \cup L_e = Lab$; wrt definition 1, $\lambda = \lambda_v \cup \lambda_e$)
– P_v:Set of vertex properties ($P_v \subset R$), $\mu_v : V \times P_v \to S$ assigns a value to each vertex property
– P_e: Set of edge properties ($P_e \subset R$), $\mu_e : E \times P_e \to S$ assigns a value to each edge property ($P_v \neq P_e$; in the context of definition 1, $\mu = \mu_v \cup \mu_e$)

Figure 1, presents a visualization of the *Apache TinkerPop* modern crew graph[4]. We use this graph as a running example throughout this paper.

2.2 Graph Pattern Matching

Graph Pattern Matching (GPM) is a computational task consisting of matching graph patterns (P) against a graph (G, ref. def. 1). Graph

[4] TinkerPop Modern Crew property graph (http://tinkerpop.apache.org/docs/3.2.3/reference/#intro)

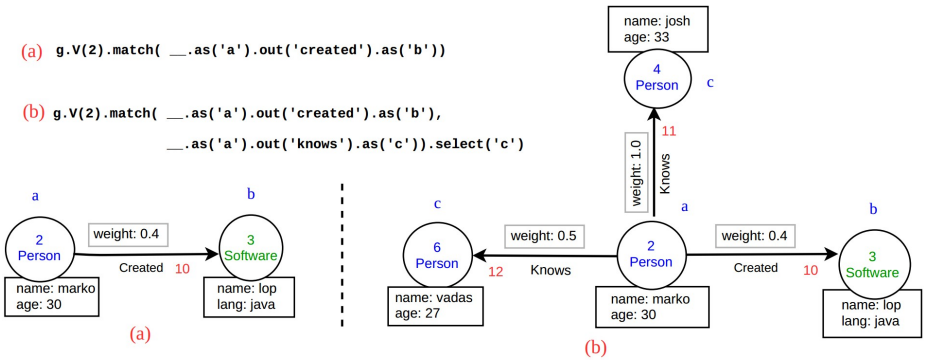

Fig. 2. We illustrate the notion of a sample, (a) basic graph pattern and (b) complex graph pattern, as a gremlin traversal over the graph G as shown in Figure 1.

databases perform GPM for querying a variety of data models such as RDF, Property Graphs, edge-labelled graphs, etc., and many works address and analyze its solvability, such as [1,8,11,13,27]. Various graph query languages have been implemented for querying these data models, that are of imperative and declarative nature, such as:

- The SPARQL[5] query language for RDF triple stores (declarative),
- Neo4J's native query language CYPHER[6] (declarative),
- Apache TinkerPop's graph traversal language and machine *Gremlin*[7] (a functional language traditionally imperative, but also offers a declarative construct).

GPM queries can represented by two types of graph patterns [1], basic graph patterns (BGP) and complex graph patterns (CGP), which add operations such as projections, unions, etc. to BGPs (cf. Figure 2).

Example 1. For the graph pattern (say P) as shown in Figure 2(b), as per the Definition 1 of property graphs, we have that P = {V, E, λ, μ}, where Figure 3 demonstrates the values of its components.

A GPM is evaluated by *matching*, a sub-graph pattern over a graph G. Matching has been formally defined in various texts and we summarise a formal definition in our context which closely follows the definition provided by [8,1].

[5] https://www.w3.org/TR/rdf-sparql-query/
[6] https://neo4j.com/developer/cypher-query-language/
[7] https://tinkerpop.apache.org/

```
V = {2,3,4,6};E = {(2,10,3),(2,11,4),(2,12,6)}
Pv = {name,age} ;Pe = {weight} Lv = {Software,Person} ;Le = {Created,Knows}

μ(2,name) = "marko";   μ(2,age) = "30"; μ(3,name) = "lop";    μ(3,lang) = "java";
μ(4,name) = "josh";    μ(4,age) = "33"; μ(6,name) = "vadas";  μ(6,age) = "27"

λ(2) = "Person";      λ(3) = "Software";  λ(4) = "Person";  λ(6) = "Person";
λ(10) = "Created";    λ(11) = "Knows";    λ(12) = "Knows"
```

Fig. 3. Demonstrating the components of the CGP shown in Figure 2(b).

Definition 2 (Match of a graph pattern). *A graph pattern* $P = (V_p, E_p, \lambda_p, \mu_p)$*; is matching the graph* $G = (V, E, \lambda, \mu)$*, iff the following conditions are satisfied:*

1. *there exist mappings* μ_p *and* λ_p *such that, all variables are mapped to constants, and all constants are mapped to themselves (i.e.* $\lambda_p \in \lambda, \mu_p \in \mu$*),*
2. *each edge* $\acute{e} \in E_p$ *in* P *is mapped to an edge* $e \in E$ *in* G*, each vertex* $\acute{v} \in V_p$ *in* P *is mapped to a vertex* $v \in V$ *in* G*, and*
3. *the structure of* P *is preserved in* G *(i.e.* P *is a sub-graph of* G*)* □

We discuss *matching* graph patterns, over a graph G, in the context of Gremlin traversal language in Section 3.2.

Example 2. We illustrate the evaluation of graph patterns (a) and (b) from Figure 2, over the graph G from Figure 1.

(a) `v[3]` (b) `v[4]`[8] `v[6]`

In Gremlin, GPM is performed by traversing[9] over a graph G. A traversal t over G derives paths of arbitrary length. Therefore, a GPM query in Gremlin can be perceived as a path traversal. Rodriguez et al. [18] define a *path* as:

Definition 3 (Path). *A path* p *is a sequence or a string, where* $p \in E^*$ *and* $E \subset (V \times L_e \times V)$[10]*. A path allows for repeated edges and the length of a path is denoted by* $||p||$*, which is equal to the number of edges in* p*.* □

[8] Please note that since we did not explicitly specify any values, the traversal returns the ids of the "matched" elements as a result.

[9] The act of visiting of vertices ($v \in V$) and edges ($e \in E$) in a graph in an alternating manner (in some algorithmic fashion) [17].

[10] The kleene start operation * constructs the free monoid $E^* = \bigcup_{n=0}^{\infty} E^i$. where $E^0 = \{\epsilon\}$; ϵ is the identity/empty element.

Moreover, from [17] we also know that these path queries are comprised of several atomic operations called the single-step traversals. A list of graph-specific operators have been defined in [17,18]. We outline these and other operators, in the following.

2.3 Graph Algebra Foundations

We present a consolidated summary of various graph query operators defined in the literature [3,9,13,14,18,17]. For brevity, we abstain from dwelling into rigorous formal definitions and underlying proofs and refer the interested reader to the respective articles.

Unary operators

Projection $(\pi_{a,b,..}) : R \cup S \rightarrow \Sigma^*$: operator projects values of a specific set of variables $a, b, .., n$ (i.e. keys and elements), from the solution of a matched input graph pattern P, against the graph G. Moreover, the results returned by $(\pi_{a,b})$ are not deduplicated be default, i.e. the result will contain as many possible matched values or items as the input pattern P. This operator is present in all standard graph query languages (e.g. `SELECT` in SPARQL, and `MATCH` in CYPHER).

Selection $(\exists(p))$, is analogous to the filter operator (σ), as defined in [13,1], restricts the match of a certain graph pattern P against a graph G, by imposing conditional expressions (p) e.g., inequalities and/or other traversal-specific predicates (where predicate is a proposition formula).

Binary operators

Concatenation [18] $(\circ)$: $E^* \times E^* \rightarrow E^*$: concatenates two paths (cf. Definition 3). For instance, if (i, α, j) and (j, β, k) are two edges in a graph G, then their concatenation is the new path (i, α, j, β, k); where $i, j, k \in V$ and $\alpha, \beta \in L_e$.

Union operator $(\uplus)$: $P(E^*) \times P(E^*) \rightarrow P(E^*)$: is the multiset union[11] (bag union) of two path traversals or graph patterns. For instance, {(1,2), (3,4), (3,4), (4,5)} $\uplus$ {(1,2), (3,4)} = {(1,2), (1,2), (3,4), (3,4), (3,4), (4,5)}. The results of this operator, like projection, are not deduplicated by default.

[11] Note that the domains and ranges of each of these sets are the power sets.

Join [18] ($\bowtie_\circ$): $P(E^*) \times P(E^*) \rightarrow P(E^*)$: produces the concatenative join of two sets of paths (path traversals) such that if $P, R \in P(E^*)$, then

$$P \bowtie_\circ R = \{p \circ r \mid p \in P \wedge r \in R \wedge (p = \epsilon \vee r = \epsilon \vee \gamma^+(p) = \gamma^-(r))\}^{12}$$

For instance, if $P = \{(v_1, e_1, v_2), (v_2, e_2, v_3)\}$ and $R = \{(v_2, e_2, v_3), (v_2, e_2, v_1)\}$, then,

$$P \bowtie_\circ R = \{(v_1, e_1, v_2, v_2, e_2, v_3), (v_1, e_1, v_2, v_2, e_2, v_1)\},$$

where $v_1, v_2, v_3 \in V; e_1, e_2 \in L_e$.

Left -join ($\rtimes$), *Right*-join ($\ltimes$) and the *Anti*-join ($\triangleright$) operators: these operators, are not explicitly implemented in Gremlin, unlike in other graph query languages [16]. Their results can, however, be simulated by the user, at run-time via selecting desired values of elements, vertices or edges declaring using the projection operator. For instance, an anti-join can be "computationally" achieved by `not`'ing an argument in the match step by using `.not()` Gremlin step.

General Extensions. We borrow the extended relational operators *Grouping* ($\dagger_a(\mathrm{p})$), *Sorting* ($\Re_{\Uparrow a, \Downarrow b}(\mathrm{p})$) and *Deduplication* ($\delta_{a,b,..}(\mathrm{p})$) which have been defined in [13]. A detailed illustration with formal definitions of extended operators can be found in [6].

Graph-specific Extensions. Various graph/traversal-specific operators have been defined in works such as [9,18]. Furthermore, there also exist certain application-specific extensions of algebra operators, such as the α and β operators, for graph data aggregation (used in complex graph network analysis) defined by [7]. We present graph-specific operators, some of which have been adapted from [9,13] and propose additional operators based on the algebra defined by [18,17].

– The *Get-vertices/Get-edges* nullary operators (V_g/ E_g): return the list of vertices/edges, respectively. These operators, w.r.t. Gremlin query construct, denote the start of a traversal. It is also possible to traverse from a specific vertex/edge in a graph, given their id's. Furthermore, they can be used to construct custom indexes over elements depending on user's choice.

[12] Here, (γ^-, γ^+) denote the first and last elements of a path respectively.

Ops	Operation	Operator	Gremlin Step	Step type
0	Get vertices	V_g	`g.V()`	-
	Get edges	E_g	`g.E()`	-
1	Selection	$\exists$(p)	`.where()`	Filter
	Property filter	$\sigma^{a}_{condition}$(p)	`.has()/.values()`	Filter
	Projection	$\Pi_{a,b,...}$(p)	`.select()`	Map
	De-duplication	$\delta_{a,b,..}$(p)	`.dedup()`	Filter
	Restriction	λ^{s}_{l}(p)	`.limit()`	Filter
	Sorting	$\Re_{\Uparrow a,\Downarrow b}$(p)	`.order().by()`	Map
	Grouping	$\dagger_{a}$(p)	`.group().by()`	Map/SideEffect
	Traverse (out/in)	$\updownarrow^{v2}_{v1}$[e](p)	`.out()/.in()`	FlatMap
2	Join	p $\bowtie_{\circ}$ r	`.and()`	Filter
	Union	p $\uplus$ r	`.union()`	Branch

Table 1. A consolidated list of relational algebra graph operators with their corresponding instruction steps in Gremlin traversal language.

- The *Traverse operator* ($\updownarrow^{v2}_{v1}$[e](p)): $P(V \cup E) \times \Sigma^* \rightarrow P(V \cup E)$: is an adapted version, analogues to the *expand-both* operator defined by [9]. The traverse operator represents the traversing over the graph operation (traversing *in* $\downarrow$ or *out* $\uparrow$ from a vertex ($v1$) to an adjacent vertex ($v2$) given the edge label `[e]`, where $(v1, v2) \in V, e \in L_e$).
- The *Property filter* operator ($\sigma^{v/e}_{condition}$(p)): $P(V \cup E) \times S \rightarrow \Sigma^*$: is a binary operator which: *(i)* filters the values of selected element (vertex/edge), if a *condition* is declared, *(ii)* otherwise, it simply returns the value of the element's property.
- The *Restriction* unary operator (λ^{s}_{l}(p)) is an adaptation of [12], which we borrow from [13]. It takes a list as input and returns the top s values, skipping specified l values. It is analogous to the LIMIT and OFFSET modifier keyword pair in SPARQL.

3 The Gremlin Graph Traversal Language and Machine

Gremlin is the query language of Apache TinkerPop[13] graph computing framework. Gremlin is system agnostic, and enables both – pattern matching (declarative) and graph traversal (imperative) style of querying over graphs.

The Machine. Theoretically, a set of traversers in T move (traverse) over a graph G (property graph, cf. Section 2.1) according to the instruc-

[13] Gremlin: Apache TinkerPop's graph traversal language and machine (https://tinkerpop.apache.org/)

tion in (Ψ), and this computation is said to have completed when there are either:

1. no more existing traversers (t), or
2. no more existing instructions (ψ) that are referenced by the traversers (i.e. program has halted).

Result of the computation being either a null/empty set (i.e. former case) or the multiset union of the graph locations (vertices, edges, labels, properties, etc.) of the halted traversers which they reference. Rodriguez et al. [16] formally define the operation of a traverser t as follows:

$$G \leftarrow {}_{\mu}\frac{t \in T}{\{\beta, \varsigma\}}{}_{\psi} \rightarrow \Psi \tag{1}$$

where, μ: $T \rightarrow U$ is a mapping from the traverser to its location in G; ψ: $T \rightarrow \Psi$ maps a traverser to a step in Ψ; β: $T \rightarrow \mathbb{N}$ maps a traverser to its bulk[14]; ς: $T \rightarrow U$ maps a traverser to its sack (local variable of a traverser) value.

The Traversal. A Gremlin graph traversal can be represented in any host language that supports function composition and function nesting. These steps are are either of:

1. *Linear motif* - $f \circ g \circ h$, where the traversal is a linear chain of steps; or
2. *Nested motif* - $f \circ (g \circ h)$ where, the nested traversal $g \circ h$ is passed as an argument to step f [16].

A step ($f \in \Psi$) can be, defined as $f : A^{\star} \rightarrow B^{\star}$[15]. Where, f maps a set of traversers of type A (located at objects of A) to a set of traversers of type B (located at objects of B). Given that Gremlin is a language and a virtual machine, it is possible to design another traversal language that compiles to the Gremlin traversal machine (analogous to how Scala compiles to the JVM). As a result, there exists various Gremlin dialects such as Gremlin-Groovy, Gremlin-Python, etc.

A Gremlin traversal (Ψ) can be compiled down to a collection of steps or instructions which form the Gremlin instruction set. The Gremlin instruction set comprises approximately 30 steps which are sufficient to provide general purpose computing and for expressing the common motifs of

[14] The bulk of a traverser is number of equivalent traversers a particular traverser represents.

[15] The Kleene star notation ($A^{\star}, B^{\star}$) denotes that multiple traversers can be in the same element (A,B).

any graph traversal query, as highlighted by [16]. However, in a majority of cases only around 10 of these instructions are sufficient to address the most common information needs (i.e. for graph pattern matching and traversal).

3.1 Graph Pattern Matching Queries in Gremlin

Gremlin provides the GPM construct, analogous to SPARQL [3,14,19][16], using the `Match()`-step. This enables the user to represent the query using multiple individual connected or disconnected graph patterns. Each of these graph patterns can be perceived as a simple path traversal, to-and-from a specific source and destination, over the graph.

Each traversal is a path query starting at a particular source (A) and terminating at a destination (B) by visiting vertices (v ∈ V) and edges (e ∈ E (V × V)) in an alternating fashion (i.e. referred to as traversing [17]). Each path query is composed of one or more single-step traversals. Through function composition and currying , it is possible to define a query of arbitrary length [17]. These path queries can be a combination of either a source, destination, labelled traversal or all of them in a varying fashion, depending on the query defined by the user.

Example 3. For instance, consider a simple path traversal to the oldest person that marko knows over the graph G as show in Figure 1. Listing 1.1 represents the gremlin query for the described traversal.

```
1 g.V().has("name","marko").out("knows").values("age").max()
```

Listing 1.1. Return the age of the oldest person marko knows

Here, `g.V()` i.e. V_g is the traverser definition bijective to V where, $\uplus_i \mu((V_g)_i)$ = V. Functionally, this query be written using function currying as:

$$max(values_{age}(out_{knows}(has_{name=marko}(V_g)))) \tag{2}$$

The terms out_{knows}, values_{age} and has_{name} are the single-step Gremlin operations/traversals. In [17], Rodriguez presents the itemisation of such single-step traversals which can be used to represent a complex path traversal. Thus, as described earlier, through functional composition and currying one can represent a graph traversal of random length. If i be the starting vertex in G, then the traversal shown in listing 1.1 can be represented as following function:

$$f(i) = max(\epsilon_{age} \circ v_{in} \circ e^{knows}_{lab+} \circ e_{out} \circ \epsilon^{marko}_{name+}) \ (i) \tag{3}$$

[16] However, Property Graphs do not encode all semantics of RDF Graphs (e.g. blank nodes.

where, $f : P(V) \rightarrow P(S)$. A detailed illustration of the single step traversals can be referred from [17].

3.2 Evaluation of `match()`-step in Gremlin

The `match()`-step[17] evaluates the input graph patterns over a graph G in a structure preserving manner binding the variables and constants to their respective values (cf. Definition 2). We denote the evaluation of a traversal t over a graph G, using the notation $[[t]]_g$ which is borrowed from [14,4]. When a `match()`-step is encountered by the Gremlin machine, it treats each graph pattern as an individual path traversal. These graph patterns are represented using `as()`-step[18] (step-modulators[19] i.e. naming variables) such as a, b, c, etc.), which typically mark the start (and end) of particular graph patterns or path traversals.

However, the order of execution of each graph pattern is up to the `match()`-step implementation, where the variables and path labels are local only to the current `match()`-step. Due to this uniqueness of the Gremlin `match()`-step it is possible to:

1. treat each graph pattern individually as a single step traversal and thus, construct composite graph patterns by joining (path-joins) each of these single step traversals;
2. combine multiple `match()`-steps for constructing complex navigational traversals (i.e. multi-hop queries), where each composite graph pattern (from a particular `match()`-step) can be joined using the concatenative join (ref. Section 2.3).

For instance, consider the GPM gremlin query as shown in Listing 1.2.

```
g.V().match(
        __.as('a').out('created').as('b'),
        __.as('b').has('name', 'lop'),
        __.as('b').in('created').as('c'),
        __.as('c').has('age', 30)).select('a','c').by('name')
```

Listing 1.2. This traversal returns the names of people who created a project named 'lop' that was also created by someone who is 30 years old.

Each of the comprising four graph patterns (traversals) of the query (listing 1.2), can be individually represented using the curried functional notation as described in Equation 2. Thus,

[17] http://tinkerpop.apache.org/docs/3.2.3/reference/#match-step

[18] Meaningful names can be used as variable names for enhancing query readability.

[19] Rodriguez et al. [17] refer to step modulators as 'syntactic sugar' that reduce the complexity of a step's arguments by modifying the previous step.

$$f(i) = (e^{created}_{lab+}) \circ e_{out}\ \ (i);\ \ g(i) = (\epsilon^{lop}_{name+} \circ v_{in})\ \ (i); \tag{4}$$

$$h(i) = (e^{created}_{lab+}) \circ e_{in}\ \ (i);\ \ j(i) = (\epsilon^{30}_{age+} \circ v_{in})\ \ (i) \tag{5}$$

The input arguments of the `match()`-step are the set of graph patterns defined above in equations 4,5, which form a composite graph pattern (the final traversal (Ψ)). At run-time, when a traverser enters `match()`-step, it propagates through each of these patterns guaranteeing that, for each graph pattern, all the prefix and postfix variables (i.e. "a", "b", etc) are *binded* with their labelled path values. It is only then allowed to exit, having satisfied this condition. In simple words, though each of these graph patterns is evaluated individually, it is made sure that at run-time, the overall structure of the composite graph pattern is preserved by mapping the path labels to declared variables.

For instance, in the query (ref. listing 1.2), the starting vertex of $g(i)$ labelled as "b" which is the terminal vertex of $f(i)$, similarly for $h(i)$ and $j(i)$ with vertex labelled as "c". It is therefore necessary, to keep a track of the current location of a traverser in the graph, to preserve traversal structure. This is achieved in Gremlin by `match()` and `bind()` functions respectively, which we outline next.

The evaluation of an input graph pattern/traversal in Gremlin is taken care by two functions:

1. the recursively defined `match()` function- which evaluates each constituting graph pattern and keeps a track of the traversers location in the graph (i.e. path history), and,
2. the `bind()` function- which maps the declared variables (elements and keys) to their respective values.

Using equations (4, 5) (curried functional form of path traversals) in the recursive definition of `match()` by [16], we have:

$$[[t]]_g = \begin{cases} [[bind_b(\mathtt{f}(t_{\Delta_a(t)} \wedge \Delta_{m1}))]]_g & : \Delta_a \neq \phi = \Delta_{m1} \\ [[\mathtt{g}(t_{\Delta_b(t)} \wedge \Delta_{m2})]]_g & : \Delta_b \neq \phi = \Delta_{m2} \\ [[bind_c(\mathtt{h}(t_{\Delta_b(t)} \wedge \Delta_{m3}))]]_g & : \Delta_b \neq \phi = \Delta_{m3} \\ [[\mathtt{j}(t_{\Delta_a(t)} \wedge \Delta_{m4})]]_g & : \Delta_c \neq \phi = \Delta_{m4} \\ t & : \text{otherwise,} \end{cases} \tag{6}$$

where, t_{Δ_a}(t) is the labelled path of traverser t. A path (ref. definition 3) is labelled "a" via the step-modulator `.as()`, of the traverser in the current traversal (Ψ); Δ_{m1}, Δ_{m2}, Δ_{m3} are hidden path labels which are appended

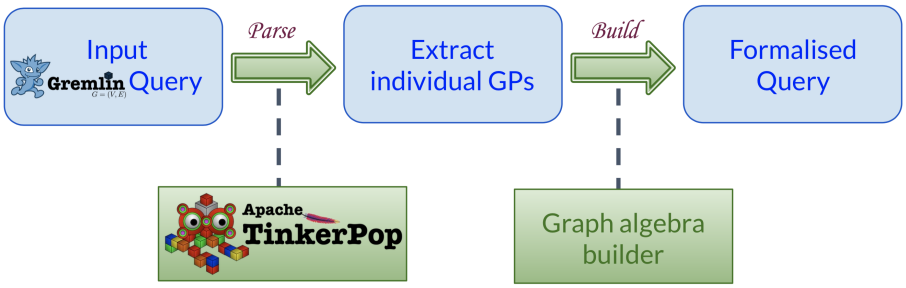

Fig. 4. Conceptual architecture for formalizing a Gremlin traversal using graph relation algebra.

to the traversers labelled path for ensuring that each pattern is executed only once; and $\text{bind}_x(\text{t})$ is defined as:

$$bind_x(t) = \begin{cases} t_{\Delta_x}(t) = \mu(t) & : \Delta_x(t) = \phi \\ t & : \Delta_x(t) = \mu'(t) \\ \phi & : \text{otherwise.} \end{cases} \tag{7}$$

where μ': T $\rightarrow$ U, is a function that maps a traverser to its current location in the graph G (e.g., v $\in$ V, V $\in$ U) [16]. It (μ') can be perceived analogues to μ defined in definition 1, which maps elements and keys to values in G, however in later case the value is the location of a traverser in G.

4 Mapping Gremlin GPM traversals to Graph Algebra

In this section, we present a mapping algorithm for encoding a given Gremlin pattern-matching traversal, relational graph algebra. Figure 4, describes the conceptual architecture for formalizing Gremlin traversals. We follow a bottom-up approach in order to construct the relational graph algebra based on the traversal.

4.1 Mapping Gremlin traversals to Graph Algebra

1. The input query is parsed and its constituent individual graph patterns are extracted from the `match()`-step and the optional `where()` step. `/* Parse step`
2. For each single graph patterns (single path traversals) in the query, we first construct the curried functional form (ref. equation 2).

3. We then map the get-vertices/get-edges operator for the encountered g.V()/g.E() step (i.e. to the first graph pattern) respectively. `/* Steps 2-12 are the Build steps`
4. Append a *traverse-operator* to all the respective in-coming and out-going edge traversals for each, that appear inside the `match()`-step.
5. Append a *property-filter* operator to all the respective `has()` and `values()` steps based on the `match()`-step.
6. Multiple `match()` steps can be connected processed using the concatenative join operator.
7. Append a *selection* operator, if the `match()` step is succeeded by a `where()` step (this is an optional in gremlin queries).
8. Append a *projection* operator, if `select()`-step is declared with a `match()` or `where()` step.
9. Append a *deduplication* operator, based on whether the `dedup()` step is declared after the `select()` step.
10. Append a *sorting* operator, if the `order()` step with an optional `by()` modifier is declared after the `select()` step.
11. Append a *grouping* operator, if the `group()` step with an optional `by()` modifier is declared after the `select()` step.
12. Map the *union* operator if the query contains a `union()`-step[20]. *Union* is technically a binary operator, however, a union of multiple patterns can be constructed using a left deep join tree representation.

Next we present three examples of gremlin traversals which have been formalised using the proposed graph relational algebra.

Example Query 1. The sample query as shown in listing 1.2, can be formalized as:

$$\dagger_{name}\left(\Pi_{a,c}\left(\underbrace{[\![\sigma^{c}_{age=30}\downarrow^{c}_{b}[created]\sigma^{b}_{name=lop}\uparrow^{b}_{a}[created](V_g)]\!]_g}_{\text{t}}\right)\right) \quad (8)$$

Example Query 2. Consider the following gremlin traversal shown in listing 1.3 below:

```
g.V().match(
    __.as('a').hasLabel('person').values('age').as('b')).
  select('b').order().by(asc)
```

Listing 1.3. This traversal returns the list all the persons in the ascending order of the age.

[20] It is not a common practice to use a union()-step in Gremlin GPM traversals, as multiple `match()`-steps in conjunction with `where()`-steps can be used as per required (the ad infinitium style of traversing [16]).

The gremlin traversal shown above (Listing 1.3) can be formalized as follows:

$$\Re_{\Uparrow_b}\left(\Pi_b[\![\underbrace{\sigma^b_{age}\sigma^a_{label=person}(V_g)}_{\text{t}}]\!]_g\right) \tag{9}$$

Example Query 3. Consider the following gremlin traversal shown in listing 1.4 below,

```
g.V().union(
  __.match( __.as('a').out('created').as('c')),
  __.match( __.as('b').out('created').as('c'))).select('a','c')
```

Listing 1.4. This traversal returns the list of all the people who have collaboratively created a software.

The gremlin traversal shown above (Listing 1.4) can be formalized as follows:

$$\Pi_{b,c}\left(\underbrace{[\![\uparrow^c_a [created](V_g)]\!]_g \uplus [\![\uparrow^c_b [created](V_g)]\!]_g}_{t = t_1 \uplus t_2}\right) \tag{10}$$

Optimizations. The Gremlin graph traversal machine inherently offers a wide variety of machine and query level optimizations as traversal strategies including (i) query rewriting (decoration), (ii) traversal optimization, (iii) vendor optimization (using byte-code for gremlin language variants), (iv) finalization, and (v) verification. We will not go into the specific details, rather we point the interested reader to (Section 4, page 6-7 of [16]).

Limitations. We do not cover the complete Gremlin language, as clarified earlier, we strictly focus on formalizing the GPM (declarative) construct. In the declarative construct of Gremlin, in this work, we only focus on covering the core functionality of pattern-based graph traversing. We do not formalize mutating and specific graph-based task (such as, centrality, eccentricity, etc) traversals and data manipulation operators, e.g., `addVertex()`, `addEdge()`, `addProperty()`, `aggregate()` operators.

5 Related Work

This section summarizes work towards formalizing query algebras for different data models and corresponding query languages.

Property graphs. The Property Graph data model is one of the popular graph data models that provides a rich set of features for the user to model domain-specific real world data. Various query languages have been proposed over the years for querying over PGs. The *Cypher* query language[21], native to the Neo4J PG database, is a high-level declarative pattern matching-based graph query language. The developers of Neo4J & Cypher strive at standardizing Cypher by providing open formal specification via the OpenCypher[22] project [20]. One of the limitations of Cypher is that it misses certain graph querying functionality such as the support for regular path queries and graph construction. *PGQL* [27], is an SQL-like syntax based graph query language for the PG data model. Albeit being able to overcome the limitations of Cypher, and lure the SQL community with its SQL-like syntax support, PGQL lacks standardization and support by database technology vendors.

RDF: The Resource Description Framework (RDF), is another graph data model, popular in the semantic web domain. In RDF the data (i.e., entity descriptions) are stored as triples, similar to the node-edge formalism in PGs. SPARQL [15], the query language for RDF triple stores, is a Cypher-like declarative GPM query language for querying RDF graphs. SPARQL is a W3C standard and its query algebra has been formally described in works such as [14,4]. Moreover, multiset semantics have also been formalized by [3,19].

SQL: Relational databases cater rather limited support for executing graph queries. However, certain databases such as PostgreSQL allow the execution of recursive queries. The SAP HANA Graph Scale-Out Extension prototype (GSE), using a SQL-type language proposed by [11], supports modelling high level graph queries.

6 Conclusion and Outlook

In this extended paper, we presented the initial efforts on formalizing GPM traversals, which is a subset of the Gremlin traversal language and machine. Since, Gremlin is both a query language and a machine, it enables the graph (pattern-match) traversals to be represented in any formal query language, given that it supports function composition and nesting. Our current work provides a theoretical foundation to leverage

[21] https://neo4j.com/developer/cypher-query-language/
[22] http://www.opencypher.org/

this advantage of Gremlin for querying graphs on various graph engines. Furthermore, our current work lays the foundation for supporting query interoperability by allowing translation of SPARQL queries to Gremlin GPM traversals [23,24], using the proposed mapping, in order to: *(i)* leverage the advantage of using graph traversal for query property graphs, and *(ii)* enable the use of SPARQL query language (the defacto standard of RDF stores) on top of other OLAP-based graph processors and OLTP-based graph engines [23,24]. This enables the semantic web community to reap best of both worlds, i.e., accessing a plethora of graph data management systems without the obligation of embracing a new graph query language. As the near future work, we aim to address the limitations pointed out in Section 4.1.

Acknowledgement

This work is supported by the funding received from EU-H2020 Framework Programme under grant agreement No. 780732 (BOOST4.0)

References

1. R. Angles, M. Arenas, et al. Foundations of modern graph query languages. *arXiv preprint arXiv:1610.06264*, 2016.
2. R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma. The linked data benchmark council: a graph and rdf industry benchmarking effort. *ACM SIGMOD Record*, 43(1):27–31, 2014.
3. R. Angles et al. The multiset semantics of SPARQL patterns. In *The Semantic Web - ISWC -15th International Semantic Web Conference, Kobe, Japan*, pages 20–36, 2016.
4. R. Angles and C. Gutierrez. The expressive power of sparql. In *International Semantic Web Conference*, pages 114–129. Springer, 2008.
5. R. Angles, H. Thakkar, and D. Tomaszuk. RDF and property graphs interoperability: Status and issues. In *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019.*, 2019.
6. H. Garcia-Molina, J. D. Ullman, et al. *Database systems: the complete book*. Pearson Education India, 2009.
7. L. Gomes-Jr, B. Amann, et al. Beta-algebra: Towards a relational algebra for graph analysis, 2015.
8. A. Gubichev and M. Then. Graph pattern matching: Do we have to reinvent the wheel? In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–7. ACM, 2014.

9. J. Hölsch and M. Grossniklaus. An algebra and equivalences to transform graph patterns in neo4j. In *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*, 2016.
10. Y. Keswani, H. Thakkar, M. Dubey, J. Lehmann, and S. Auer. The litmus test: Benchmarking rdf and graph data management systems. In *Proceedings of 13th International Conference on Semantic Systems-Poster & Demos*, 2017.
11. C. Krause et al. An sql-based query language and engine for graph pattern matching. In *International Conference on Graph Transformation*, pages 153–169. Springer, 2016.
12. C. Li, K. C.-C. Chang, et al. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 131–142. ACM, 2005.
13. G. S. J. Marton. Formalizing opencypher graph queries in relational algebra. *Published online on FTSRG archive*, 2017.
14. J. Pérez, M. Arenas, et al. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.
15. E. Prud et al. Sparql query language for rdf. *Citeulike Online Archive*, 2006.
16. M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA*, pages 1–10, 2015.
17. M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph Data Management: Techniques and Applications.*, pages 29–46. IGI Global, 2011.
18. M. A. Rodriguez and P. Neubauer. A path algebra for multi-relational graphs. In *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011*, pages 128–131, 2011.
19. M. Schmidt, M. Meier, et al. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, pages 4–33. ACM, 2010.
20. G. Szárnyas et al. opencypher specification. *Online FTSRG archive*, 2017.
21. H. Thakkar. Towards an open extensible framework for empirical benchmarking of data management solutions: LITMUS. In *European Semantic Web Conference*, pages 256–266. Springer, 2017.
22. H. Thakkar, Y. Keswani, M. Dubey, et al. Trying not to die benchmarking: Orchestrating RDF and graph data management solution benchmarks using LITMUS. In *Proceedings of the 13th International Conference on Semantic Systems, SEMANTICS 2017, Amsterdam, The Netherlands, September 11-14, 2017*, 2017.
23. H. Thakkar, D. Punjani, et al. A stitch in time saves nine–SPARQL querying of property graphs using gremlin traversals. *arXiv:1801.02911*, 2018.
24. H. Thakkar, D. Punjani, J. Lehmann, and S. Auer. Two for one: Querying property graph databases using SPARQL via gremlinator. In *Proc. of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–5. ACM, 2018.
25. H. Thakkar, D. Punjani, M.-E. Vidal, and S. Auer. Towards an integrated graph algebra for graph pattern matching with gremlin. In *Proceedings of the 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I*, pages 81–91. Springer, 2017.
26. C. Unger, A. N. Ngomo, et al. 6th open challenge on question answering over linked data (QALD-6). In *Semantic Web Challenges - Third SemWebEval Challenge at ESWC 2016, Heraklion, Crete, Greece*, pages 171–177, 2016.

27. O. van Rest, S. Hong, et al. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2016.