A

Foundations of Modern Query Languages for Graph Databases[1]

RENZO ANGLES, Universidad de Talca & Center for Semantic Web Research
MARCELO ARENAS, Pontificia Universidad Católica de Chile & Center for Semantic Web Research
PABLO BARCELÓ, DCC, Universidad de Chile & Center for Semantic Web Research
AIDAN HOGAN, DCC, Universidad de Chile & Center for Semantic Web Research
JUAN REUTTER, Pontificia Universidad Católica de Chile & Center for Semantic Web Research
DOMAGOJ VRGOČ, Pontificia Universidad Católica de Chile & Center for Semantic Web Research

We survey foundational features underlying modern graph query languages. We first discuss two popular graph data models: *edge-labelled graphs*, where nodes are connected by directed, labelled edges; and *property graphs*, where nodes and edges can further have attributes. Next we discuss the two most fundamental graph querying functionalities: *graph patterns* and *navigational expressions*. We start with graph patterns, in which a graph-structured query is matched against the data. Thereafter we discuss navigational expressions, in which patterns can be matched recursively against the graph to navigate paths of arbitrary length; we give an overview of what kinds of expressions have been proposed, and how they can be combined with graph patterns. We also discuss several semantics under which queries using the previous features can be evaluated, what effects the selection of features and semantics has on complexity, and offer examples of such features in three modern languages that are used to query graphs: SPARQL, Cypher and Gremlin. We conclude by discussing the importance of formalisation for graph query languages; a summary of what is known about SPARQL, Cypher and Gremlin in terms of expressivity and complexity; and an outline of possible future directions for the area.

CCS Concepts: •**Information systems** → **Query languages;** •**Theory of computation** → **Database query languages (principles);**

Additional Key Words and Phrases: Property graphs, graph databases, query languages, graph patterns, navigation, aggregation

ACM Reference Format:
ACM V, N, Article A (January YYYY), 42 pages.
DOI: 0000001.0000001

1. INTRODUCTION

The last decade has seen a resurgence of interest in *graph databases*, wherein entities from the domain of interest are represented by *nodes* and relationships between them by *edges*. Part of this resurgence stems from the growing realisation that there are a variety of domains for which graph databases offer a more intuitive conceptualisation than their more well-established relational cousins. For example, one can view a social network as a graph of people who know each other. One may likewise view transport networks, biological pathways, citation networks, and so on, as a graph. Although graphs can still be (and sometimes still are) stored in relational databases, the choice to use a graph database for certain domains has significant benefits in terms of querying, where the emphasis shifts from joining various tables to specifying graph patterns and navigational patterns between nodes that may span arbitrary-length paths. To support these new types of queries, a variety of graph database engines [Erling 2012; Thompson et al. 2014; The Neo4j Team 2016], graph data models [Harris and Seaborne 2013; The Neo4j Team 2016] and graph query languages [The Neo4j Team 2016; Harris and Seaborne 2013; Apache TinkerPop 2017] have been released over the past few years.

Scope. Our goal in this survey is to give an in-depth discussion of the main conceptual features found in modern graph query languages, as both supported by graph database engines, and studied in the theoretical literature. By organising our survey

[1]Work funded by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004.

at the level of query features, rather than languages, we provide a foundational introduction to the area, which helps to understand, and even define, individual query languages as the composition of features. We consider two high-level categories of query features: graph patterns and path expressions. These features collectively form the core of a variety of modern graph query languages [The Neo4j Team 2016; Harris and Seaborne 2013; van Rest et al. 2016], and form the core of what has been studied in the theoretical literature [Wood 2012; Barceló 2013].

After introducing and defining the graph query features in each category, we list various semantics under which such features can be evaluated, provide examples of how such features are applied in a selection of modern query languages, discuss the computational complexity of key problems underlying such features, and present some of their most important extensions as implemented in modern graph database engines.

We wrap up by drawing all of the foundational discussion together into a summary of the types of features we have covered, how these features can be used to understand the complexity and expressivity of modern query languages, the importance of formalisation for such languages, the key challenges underlying their implementation and optimisation in practical engines, and possible ways in which they might evolve.

Survey structure. The survey is structured as follows:

— We first discuss two graph *data models* in Section 2: *edge-labelled graphs*, which is the foundational model considered in the graph database literature; and *property graphs*, which is a model commonly employed in practice, where nodes and edges in labelled graphs can be annotated with additional meta-information.
— In Section 3, we discuss *graph patterns*, where a graph-structured query is matched against the graph database. We also discuss the extension of such graph patterns with additional operators, such as projection, difference, union, etc.
— Section 4 then introduces *navigational expressions*, which, unlike graph patterns, can match paths of arbitrary length.
— In Section 5 we present our final remarks.

Online Appendix. An online appendix for this paper contains several aspects of graph query languages that, while not mainstream, have enjoyed some coverage in the research literature and in practical systems. These include an alternative semantics for matching patterns to graphs; extensions of path queries to recursive patterns and Datalog; additional query features such as *aggregation*, *path unwinding* and *graph-to-graph* queries; as well as further extensions that can be considered.

Proviso. Throughout the survey, following the conventions of theoretical papers, we will use the phrase “graph database” to refer to a specific data model or an instance of that data model. We will use the phrase “graph database engine” to specify an implementation for executing queries over graph databases.

Intended audience. The main ambition of this survey is to bridge theory and practice, relating theoretical notions of querying graphs to three modern query languages that are popular in practice. The survey is thus primarily aimed at both theoretical and applied researchers interested in graph databases. For a theory-oriented researcher, the survey outlines a practical context for proposals in the theoretical literature, providing concrete examples of how practical languages instantiate or relate to theoretical proposals, discussing choices of semantics, highlighting aspects of such languages not yet well understood in theory, etc. For a practice-oriented researcher, the survey shows how the core of various graph query languages can be understood and compared from a more foundational perspective, the possible semantics that can be chosen, the effects on complexity and the practicality of a language by changing certain features

and/or semantics, etc. Aside from researchers, practitioners – i.e., developers, database administrators, engineers, consultants – may also be interested in this survey, particularly those involved in the development of graph database engines or query languages.

To keep the paper accessible to a broad audience, we keep formal definitions only for core notions where it is important to be precise. Throughout the survey, we provide a wide variety of examples, including examples in three concrete query languages: SPARQL, Cypher and Gremlin.

Previous surveys. A number of surveys have been published in recent years in the area of graph databases. Angles and Gutiérrez [2008] provide a survey of graph database models. More recently, Angles [2012] presents a systematic analysis of the functionalities of current graph database engines. Neither of these surveys covers querying graph databases in depth, rather focusing on models and engines.

Wood [2012] and Barceló [2013] study several graph query languages from a theoretical point of view, focusing on their expressive power and the computational complexity of associated problems. Given the theoretical focus of both surveys, neither covers practical aspects of modern graph query languages in detail.

Particular aspects of graph querying have also been surveyed; for example, works by Bunke [2000], Gallagher [2006], Riesen et al. [2010], Livi and Rizzi [2013], and Yan et al. [2016] deal with particular aspects of graph pattern matching, while Yu and Cheng [2010] concentrate on graph reachability queries. Again, however, all such works have a narrower focus than our survey.

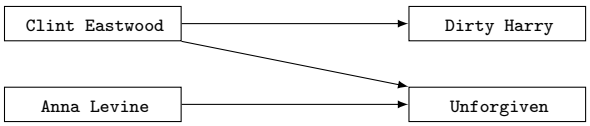
Our survey complements these previous works in two novel aspects:

(1) Instead of surveying the myriad of different graph data models available, we build our presentation in terms of two popular such data models; namely, edge-labelled and property graphs. In spite of their simplicity, these models are flexible enough to express most practical graph database scenarios. In addition, the most fundamental issues related to querying graphs are already present for these models.
(2) Though we discuss semantics and complexity, we do not focus only on the theoretical aspects of graph query languages. Instead, we identify and explain in detail the basic features that appear in such languages, providing examples of how they are applied in a selection of practical query languages. In summary, our paper bridges the theory and practice of graph query languages in a novel manner; as previously discussed, our survey thus targets a broader audience than previous works.

Specific novel aspects of this survey include a new formalisation of the property graph model; discussion of how this model can be understood through the lens of existing theory; comparisons of practical aspects of the SPARQL, Gremlin and Cypher query languages and the semantics they adopt; and examples of how the design of such languages influences the complexity of query evaluation.

2. GRAPH DATA MODELS

Graphs can be used to encode data whereby nodes represent objects in a domain of interest, and edges represent relationships between these objects. For instance, if a graph is used to encode data about movies, nodes may be actors and movies, and a (directed) edge from a node a to a node b may indicate that a is an actor in b. Note that the direction of an edge matters here: we want to say that an actor stars in a movie, but not vice-versa. A movie database can then be modelled using graphs as follows:



However, it is difficult to express different types of relationships in such a simple form of graph. For instance, suppose that we wish to encode that `Clint Eastwood` is also the director of `Unforgiven`. We could consider adding an edge between these nodes, thus ending up with two nodes connected in the following way:

But which edge here represents the fact that `Clint Eastwood` is the director of `Unforgiven`? And, more generally, if we have many different types of relationships between nodes, how can we distinguish between them?

Edge-labelled graphs. A simple and widely-adopted solution is the use of *edge-labelled graphs*, where we additionally assign *labels* to edges that indicate the different types of relationships in the domain being described. We can see an example in Figure 1 where `Clint Eastwood` has two relations to `Unforgiven`: one represented by the edge labelled `acts_in`, another represented by the edge labelled `directs`, and where `Anna Levine` also has an edge labelled `acts_in` to this movie.

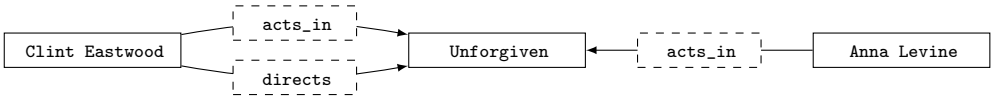


Fig. 1. An edge-labelled graph encoding basic movie information with dashed labels on edges

In the following, we formalise the notion of an *edge-labelled graph*.

Definition 2.1 (*Edge-labelled graph*). An edge-labelled graph G is a pair (V, E), where:

(1) V is a finite set of *vertices* (or *nodes*).
(2) E is a finite set of *edges*; formally, $E \subseteq V \times \mathit{Lab} \times V$ where Lab is a set of *labels*. □

Example 2.2. Letting $G = (V, E)$ denote the graph from Figure 1, the set of vertices and edges, respectively, are:

$$
\begin{aligned}
V &= \{\, \texttt{Clint Eastwood}, \texttt{Unforgiven}, \texttt{Anna Levine} \,\} \\
E &= \{\, (\texttt{Clint Eastwood}, \texttt{acts\_in}, \texttt{Unforgiven}), \\
&\quad\ (\texttt{Clint Eastwood}, \texttt{directs}, \texttt{Unforgiven}), \\
&\quad\ (\texttt{Anna Levine}, \texttt{acts\_in}, \texttt{Unforgiven}) \,\}
\end{aligned}
$$

The labels `acts_in` and `directs` are taken from the set Lab. □

Edge-labelled graphs are widely adopted in practice where, for example, they form the basis of the Resource Description Framework (RDF) standard used for encoding machine-readable content on the Web [Klyne et al. 2014]. An *RDF graph* is simply a set of triples analogous to the edges in a graph database, but with some further detailing: in the case of RDF, the set V can be partitioned into disjoint sets of IRIs, literals and blank nodes, and the set Lab is a subset of IRIs (not necessarily disjoint from V). But

for our purposes, we require no special consideration on the types of nodes,[2] and for simplicity, we consider an RDF graph as simply a special type of edge-labelled graph.

Note that the definition of an edge-labelled graph does not impose any particular restriction on the topology of graphs. For example, although Figure 1 does not contain a cycle, one can be obtained if we also add an edge labelled `directedBy` between Unforgiven and Clint Eastwood, signifying that the movie was directed by Clint Eastwood. For more involved cycles please refer to our social network example in Figure 5 below.

Although edge-labelled graphs have a simple structure, they can encode complex information. For example, when describing certain movies in a graph database, we may wish to encode that an actor has acted multiple times in the same movie under different roles. At first, this may seem incompatible with our definition of a graph database $G = (V, E)$ since E is defined as a *set* of edges: we cannot have multiple edges with the same label between the same two nodes. However, with some lateral thinking, we can model such information as an edge-labelled graph, as per Figure 2. Here we see that by using a node (rather than an edge) to represent each role played by the actor in the movie, we can not only encode cases where an actor plays multiple roles in a movie, but we can also encode additional information about the role, in this case the total on-screen time for the character in question. With this principle of using nodes to represent *n-ary relations* (where $n > 2$), it becomes feasible to encode increasingly more complex information in an edge-labelled graph, such as, for example, to encode that the same character can be portrayed by different actors, and so forth.

Property graphs. In edge-labelled graphs, we use labels to indicate the type of edge, where multiple edges may have the same type. In a similar way, we could consider labelling nodes.[3] For example, in the movie graph of Figure 1, we could label the nodes `Clint Eastwood` and `Anna Levine` as `Person`, and the node `Unforgiven` as `Movie`; we may even add multiple labels to a node, for example to label `Clint Eastwood` as `Director` and `Actor`. While this information can be represented in edge-labelled graphs – for example, as done in RDF, a new node is created for `Movie` with an edge labelled `type` extended to it from `Unforgiven` – having node labels as part of the model can offer a more direct abstraction that is easier for users to query and understand.

In the same way, it is often cumbersome to add information about the edges to an edge-labelled graph. For example, let's say that to Figure 1, we wished to add the source of information, e.g., that the `acts_in` relations were sourced from the web-site IMDb; for this, we cannot simply add edges to the graph. Instead, we would need to start again from the graph in Figure 1, and create a new type of n-ary relation with the information we need: the facts in the `acts_in` relation together with their source of information.[4] Adding new types of information to edges in an edge-labelled graph may thus require a major change to the graph's structure, entailing a significant cost.

Thus, for scenarios where various new types of meta-information may regularly need to be added to edges or nodes, the most general and widely adopted alternative is to use an extension of an edge-labelled graph called a *property graph*. This model is currently adopted by some major graph database engines, such as Neo4j [Robinson et al. 2013], and has been recently standardised by a working group of the Linked Data Benchmark Council (LDBC) formed by members of academia and industry [LDBC 2015].

[2]We will not consider the existential semantics of blank nodes nor the interpretation of datatype values nor other special vocabularies. These issues are orthogonal to our goal of introducing query features for graphs.
[3]Such graphs are often called *heterogeneous information networks*, see e.g. [Sun et al. 2011].
[4]A more generic technique involves applying "reification" where edges are represented as nodes (for a more detailed discussion see [Hernández et al. 2015]).

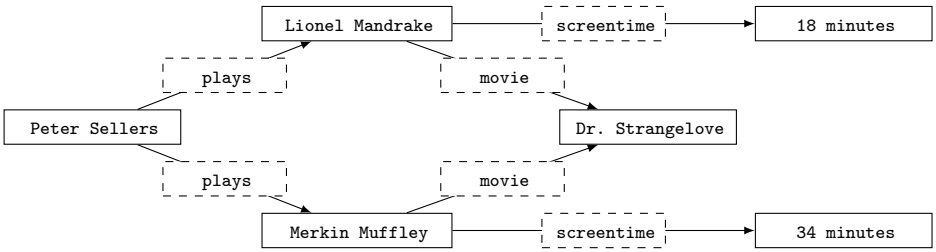


Fig. 2. An edge-labelled graph encoding information about actors that have acted in movies under different roles

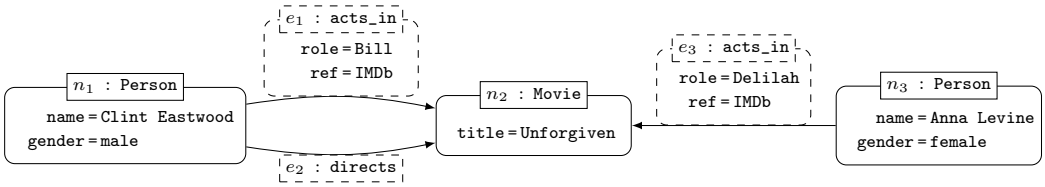


Fig. 3. A property graph with attribute values storing information about movies.

In property graphs, both edges *and* nodes can be labelled. Each edge and node is additionally associated with a unique identifier that can be used as a "hook" to associate additional meta-information – in the form of a set of property–value pairs called *attributes* – directly to that edge or node. Again, while it would be possible to instead encode attributes and labels as additional edges, in practice, such features allow one to directly annotate the graph without modifying its overall structure.

For example, in Figure 3 we show a graph for our movie database that includes labels and attributes on nodes and edges. In this figure, the attributes for a node are shown in the round rectangle below it. Thus, for example, the attributes associated to the node with identifier n_1 are *name* and *gender*, and their values are `Clint Eastwood` and `male`, respectively. On the other hand, the edge with identifier e_2 does not have any attribute. In this model, we can directly encode multiple edges (having different identifiers) with the same label between the same two nodes, and can extend the graph with additional attributes on edges without having to remodel complex relations as nodes.

We now provide a formal definition of the notion of a property graph.

Definition 2.3 (*Property graph*). A property graph G is a tuple $(V, E, \rho, \lambda, \sigma)$, where:

(1) V is a finite set of *vertices* (or *nodes*).
(2) E is a finite set of *edges* such that V and E have no elements in common.
(3) $\rho : E \to (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ indicates that e is a directed edge *from* node v_1 *to* node v_2 in G.
(4) $\lambda : (V \cup E) \to \textit{Lab}$ is a total function with *Lab* a set of labels. Intuitively, if $v \in V$ (resp., $e \in E$) and $\rho(v) = \ell$ (resp., $\rho(e) = \ell$) , then ℓ is the label of node v (resp., edge e) in G.
(5) $\sigma : (V \cup E) \times \textit{Prop} \to \textit{Val}$ is a partial function with *Prop* a finite set of properties and *Val* a set of values. Intuitively, if $v \in V$ (resp., $e \in E$), $p \in \textit{Prop}$ and $\sigma(v, p) = s$ (resp., $\sigma(e, p) = s$), then s is the value of property p for node v (resp., edge e) in the property graph G. □

Example 2.4. For the property graph G shown in Figure 3, we have that $G = (V, E, \rho, \lambda, \sigma)$, where V, E, ρ, λ, and σ are as shown in Figure 4. □

$$V = \{n_1, n_2, n_3\} \qquad E = \{e_1, e_2, e_3\}$$
$$\rho(e_1) = (n_1, n_2) \qquad \rho(e_2) = (n_1, n_2)$$
$$\rho(e_3) = (n_3, n_2)$$
$$\lambda(n_1) = \texttt{Person} \qquad \lambda(n_2) = \texttt{Movie}$$
$$\lambda(n_3) = \texttt{Person} \qquad \lambda(e_1) = \texttt{acts\_in}$$
$$\lambda(e_2) = \texttt{directs} \qquad \lambda(e_3) = \texttt{acts\_in}$$
$$\sigma(n_1, \texttt{name}) = \texttt{Clint Eastwood}$$
$$\sigma(n_1, \texttt{gender}) = \texttt{male}$$
$$\sigma(n_2, \texttt{title}) = \texttt{Unforgiven}$$
$$\sigma(n_3, \texttt{name}) = \texttt{Anna Levine}$$
$$\sigma(n_3, \texttt{gender}) = \texttt{female}$$
$$\sigma(e_1, \texttt{role}) = \texttt{Bill}$$
$$\sigma(e_1, \texttt{ref}) = \texttt{IMDb}$$
$$\sigma(e_3, \texttt{role}) = \texttt{Delilah}$$
$$\sigma(e_3, \texttt{ref}) = \texttt{IMDb}$$

Fig. 4. The components of the graph G shown in Figure 3

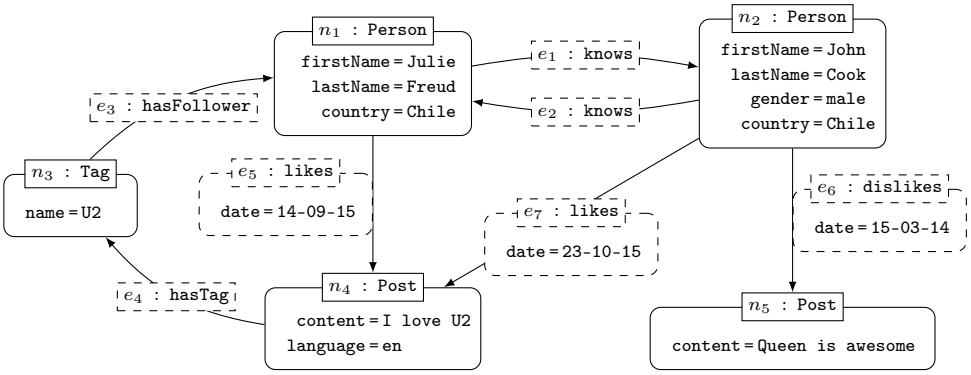


Fig. 5. A property graph storing social network data.

In our definition of a property graph, each node and edge is associated with a single label, and at most one value for each attribute property. In some applications, it may be useful to have multiple values in these positions. We could thus consider a variant of property graphs, which we call *multi-valued property graphs*, to allow multiple labels and multi-valued attribute properties within the property graph model: in Definition 2.3, the mapping λ would then return a set of labels and σ would return a set of values. In practice, engines may have custom policies; for example, Neo4j [The Neo4j Team 2016] – a popular engine implementing the property graph model that we will introduce later – allows only one label on each edge, multiple labels on nodes, and one value on each attribute property (albeit potentially a list). In any case, we focus on the single-valued variant of a property graph as given in Definition 2.3; whether or not λ or σ return a single label/value or sets of labels/values is not exigent for us.

We conclude our discussion about property graphs by presenting a second real-world example of how connected data can be modelled by using this class of graphs.

Example 2.5. A property graph representation of a (fictitious) social network is shown in Figure 5. Each node is labelled either as `Person`, `Post`, or `Tag`, and each edge is labelled either as `dislikes`, `knows`, `likes`, `hasFollower` or `hasTag`. Nodes with label `Person` may have attributes for `firstName`, `lastName`, `gender` and `country`; nodes with label `Tag` may have an attribute for `name`; nodes with label `Post` may have attributes for `content` and `language`; and edges with label `dislikes` or `likes` may have an attribute for `date`. We highlight that edge-sets $\{e_1, e_2\}$, $\{e_3, e_4, e_5\}$, etc., form directed cycles. □

Per the proviso in the introduction, in the following, we refer to edge-labelled graphs and property graphs generically as *graph databases*. We refer to systems implementing such a data model as *graph database engines*.

3. GRAPH PATTERNS

A variety of practical, declarative query languages have emerged in the past ten years for interrogating instances of graph data models presented in the previous section. One of the earliest such languages to be adopted by multiple vendors – for the purposes of querying RDF graphs – was *SPARQL* (SPARQL Protocol and RDF Query Language), which was initially standardised by the W3C in 2008 [Prud'hommeaux and Seaborne 2008], with an updated version called SPARQL 1.1 published in 2013 [Harris and Seaborne 2013]. With respect to property graphs, perhaps the most well-known implementation thereof is the Neo4j engine, whose development team released a declarative query language called Cypher [The Neo4j Team 2016]. Another query language for property graphs is Gremlin [Apache TinkerPop 2017], which forms an important part of the Apache TinkerPop3 graph computing framework.[5]

Although these three query languages vary significantly in terms of style, purpose, expressivity, implementation, etc., they share a common conceptual core, which consists of two natural operations that one could imagine in the context of querying graphs: *graph pattern matching* and *graph navigation*. In this section, we focus on the former operation; navigation will be covered in detail in Section 4.

The simplest form of graph pattern is a *basic graph pattern*, which is a graph-structured query that should be matched against the graph database. Additionally, basic graph patterns can be augmented with other (relational-like) features, such as projection, union, optional and difference. These allow for refining what sorts of matches are allowed and, ultimately, what results are returned. We call basic graph patterns augmented with such features *complex graph patterns*. Graph pattern matching is then the evaluation of graph patterns over graph databases; it forms part of the conceptual core of SPARQL, Cypher and Gremlin; it has also found use in a variety of practical applications, including chemical structure analysis, machine learning, planning, semantic networks, and pattern recognition (see, e.g., [Bunke 2000; Aggarwal and Wang 2010; Ogata et al. 2000; Matono et al. 2003; Milo et al. 2002]).

We begin by introducing basic graph patterns and complex graph patterns, discuss different semantics used to evaluate them, and present concrete examples of graph patterns in SPARQL, Cypher and Gremlin. Thereafter, we make some general remarks on the complexity of graph pattern matching.

3.1. Basic graph patterns

At the core of query answering over graph databases is basic graph pattern matching.[6] Basic graph patterns (bgps) follow the same structure as the type of graph database they are intended to query but instead of only allowing constants, basic graph patterns also permit variables. In other words, a bgp for querying an edge-labelled graph is just an edge-labelled graph where variables can now appear as nodes or edge labels; a bgp for querying property graphs is just a property graph where variables can appear in place of any constant. A *match* for a bgp is a mapping from variables to constants such that when the mapping is applied to the bgp, the result is, roughly speaking, contained

[5]Although SPARQL (1.1) has been officially standardised, Cypher and Gremlin have not and are subject to change. This survey is based on Cypher/Neo4j v.3 and Gremlin/TinkerPop v.3. Issues we discuss relating to these languages may thus change in future versions. However, given that many such systems now rely on these languages, significant (non-backwards-compatible) changes to the core features covered here would incur major migration costs. In revising the recent change-logs of these languages, we informally note that the core features and semantics discussed in this survey have not changed in recent years.

[6]In the context of query answering over graphs, basic graph patterns are equivalent to *conjunctive queries* [Abiteboul et al. 1995] without projection (which will be added later).

within the original graph database. The results for a bgp are then all mappings from variables in the query to constants in the database that comprise a match.

We start with an example of a bgp for an edge-labelled graph; later we will give a more complex example involving a bgp for a property graph.

Example 3.1. Let G be the graph in Figure 1. Assume we wish to find all co-stars in this graph. We can do this by matching the bgp in Figure 6(a), which we shall call Q, against G. In Q, we use terms x_i as variables that will match any term in the database. On the other hand, `acts_in` is a constant from the set *Lab* that will only match edges with the corresponding label in the original graph. The results of evaluating the bgp Q against the graph G, which we denote as $Q(G)$, will thus be as follows:

x_1	x_2	x_3
Clint Eastwood	Anna Levine	Unforgiven
Anna Levine	Clint Eastwood	Unforgiven
Clint Eastwood	Clint Eastwood	Unforgiven
Anna Levine	Anna Levine	Unforgiven

Taking the first mapping as an example, in the original bgp, if we replace variable x_1 by `Clint Eastwood`, x_2 by `Anna Levine` and x_3 by `Unforgiven`, we get a sub-graph of the original graph database; thus we call this mapping a *match* for Q against G. The results then consist of all such valid matches. □

Though not shown in the prior example, we may also refer to specific nodes in the bgp; for example, to find the co-stars of Clint Eastwood, we could replace the variable x_1 (or x_2) with the term `Clint Eastwood`. Basic graph patterns may also contain cycles, where, for example, we could also query for co-stars who are siblings.

We now look at an example of a bgp for a property graph.

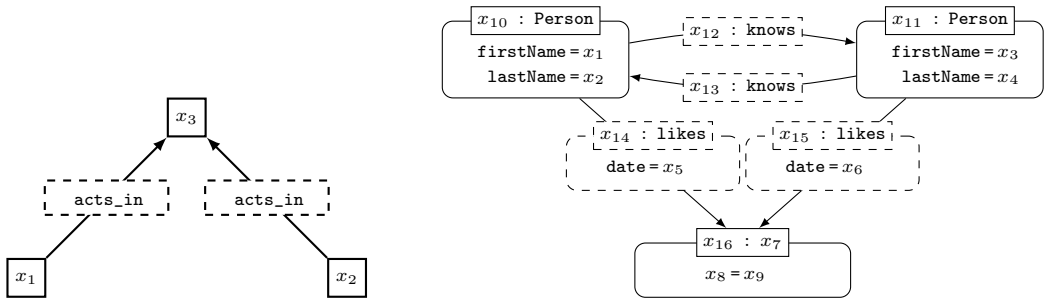
Example 3.2. Let G be the property graph in Figure 5. Assume we wish to query for things that (mutual) friends in the social network both like, where we wish to view the first and last name of the users in question, all the details of the item(s) they both like, and the date on which they both liked the item(s) in question. We can achieve this by matching the bgp in Figure 6(b), which we shall call Q, against the graph G. Again, we use terms x_i as variables that will match any term in the graph database. In this case, the results $Q(G)$ will be as follows:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	...
Julie	Freud	John	Cook	14-09-15	23-10-15	Post	content	I love U2	n_1	...
John	Cook	Julie	Freud	23-10-15	14-09-15	Post	content	I love U2	n_2	...
Julie	Freud	John	Cook	14-09-15	23-10-15	Post	language	en	n_1	...
John	Cook	Julie	Freud	23-10-15	14-09-15	Post	language	en	n_2	...

We omit the columns for variables x_{11}–x_{16} for space reasons: these variables will simply match the corresponding node ids and edge ids in a manner analogous to x_{10}. Please note that in the expression $x_8 = x_9$, the equality sign refers to a mapping from the attribute name to its value (not equality between variables).

As for the previous example, if we replace the variables in Q per any of the mappings in the results above, we find that the corresponding property graph is contained within G, where $Q(G)$ is again defined to contain all (and only) such matches. □

Definition. More formally, let us refer collectively to the sets of terms V and *Lab* from Definition 2.1 and the sets of terms V, E, *Lab*, *Prop* and *Val* from Definition 2.3 as *constants*, denoted *Const*. Let *Var* denote a set of *variables*. We could then define bgps for graph databases in relation to Definition 2.1 by allowing V and *Lab* to contain



(a) Example for an edge-labelled graph

(b) Example for a property graph

Fig. 6. Two example basic graph patterns: 6(a) applies to the graph database depicted in Figure 1 while 6(b) applies to the property graph depicted in Figure 5

variables, and likewise we could define bgps for property graphs in relation to Definition 2.3 by allowing V, E, *Lab*, *Prop* and *Val* to contain variables. For brevity, we skip repetitive definitions and instead continue with some quick examples.

Example 3.3. For the bgp Q shown in Figure 6(a), as per Definition 2.1, we can denote $Q = (V, E)$, where:

$$V = \{x_1, x_2, x_3\}\,,\; E = \{(x_1, \texttt{acts\_in}, x_3), (x_2, \texttt{acts\_in}, x_3)\}$$

In this case, $x_i \in \textit{Var}$ for $1 \leq i \leq 3$, while $\texttt{acts\_in} \in \textit{Const}$. □

Example 3.4. For the bgp Q shown in Figure 6(b), as per Definition 2.3, we can denote $Q = (V, E, \rho, \lambda, \sigma)$, where:

$$\begin{array}{lll}
V = \{x_{10}, x_{11}, x_{16}\} & E = \{x_{12}, x_{13}, x_{14}, x_{15}\} & \sigma(x_{10}, \texttt{firstName}) = x_1 \\
 & & \sigma(x_{10}, \texttt{lastName}) = x_2 \\
\rho(x_{12}) = (x_{10}, x_{11}) & \rho(x_{13}) = (x_{11}, x_{10}) & \sigma(x_{11}, \texttt{firstName}) = x_3 \\
\rho(x_{14}) = (x_{10}, x_{16}) & \rho(x_{15}) = (x_{11}, x_{16}) & \sigma(x_{11}, \texttt{lastName}) = x_4 \\
 & & \sigma(x_{14}, \texttt{date}) = x_5 \\
\lambda(x_{10}) = \texttt{Person} & \lambda(x_{12}) = \texttt{knows} & \sigma(x_{15}, \texttt{date}) = x_6 \\
\lambda(x_{11}) = \texttt{Person} & \lambda(x_{13}) = \texttt{knows} & \sigma(x_{16}, x_8) = x_9 \\
\lambda(x_{16}) = x_7 & \lambda(x_{14}) = \texttt{likes} & \sigma(x_{16}, x_8) = x_9 \\
 & \lambda(x_{15}) = \texttt{likes} &
\end{array}$$

As before, $x_i \in \textit{Var}$ for $1 \leq i \leq 16$, and all other domain terms are in *Const*. □

Evaluation. Evaluating a bgp Q against a graph database G corresponds to listing all possible *matches* of Q with respect to G (as per Examples 3.1 & 3.2). More formally, we can define a match as follows.

Definition 3.5 (*Match*). Given an edge-labelled graph $G = (V, E)$ and a bgp $Q = (V', E')$, a *match* h of Q in G is a mapping from $\textit{Const} \cup \textit{Var}$ to *Const* such that:

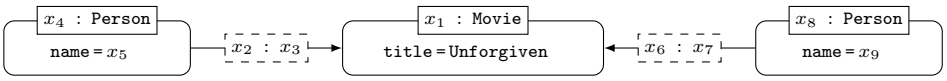
(1) for each constant $a \in \textit{Const}$, it is the case that $h(a) = a$; that is, the mapping maps constants to themselves; and
(2) for each edge $(b, l, c) \in E'$, it holds that $(h(b), h(l), h(c)) \in E$; this condition imposes that (a) each edge of Q is mapped to an edge of G, and (b) the structure of Q is preserved in its image under h in G (that is, when h is applied to all the terms in Q, the result is a sub-graph of G). □

We leave implicit the analogous definition for property graphs since the principle is the same: a mapping h maps constants to themselves and variables to constants; if the image of Q under h is contained within G, then h is a match (see Example 3.2).

In technical terms, a match h corresponds to a *homomorphism* from Q to G (see, e.g., [Barceló 2013]), whereby multiple variables in Q can map to the same term in G, as was the case in Example 3.1 where the latter two matches mapped variables x_1 and x_2 to the same term. In some cases, however, it may be desirable to require that variables map to distinct terms, where these latter two matches would be dropped; in other words it may be desirable to restrict h to be an injective (i.e., one-to-one) mapping, in which case the matching process corresponds to the well-known notion of *subgraph isomorphism* (see, e.g., [Ullmann 1976; Fan 2012]). But this may be too strict in certain applications, where, for example, it may be desirable to allow multiple label variables to match one label, but to enforce that node and/or edge ids are kept distinct (with the intuition that nodes and edges represent the structure of the graph, and labels are simply annotations on that structure). These preferences lead to different semantics for the *evaluation of a bgp Q over a graph database G*, as explained next:

(1) *Homomorphism-based semantics:* This is the unconstrained semantics: no additional restriction is imposed on the matches h of Q in G other than the base conditions from Definition 3.5. The evaluation of Q against G then consists of all possible homomorphisms from Q to G. Since homomorphism-based approach corresponds to the familiar semantics of select-from-where queries in relational databases, and since it forms the basis for the other, more restrictive semantics of bgps, it is often studied in the theoretical community (see, e.g., [Calvanese et al. 2000; Wood 2012; Barceló 2013; Barceló et al. 2014; Reutter et al. 2015a]). There are also several papers that study implementation issues related to this semantics (see, e.g., [Cheng et al. 2008; Zou et al. 2009; Fan et al. 2010b]) and it is currently used, for example, by the SPARQL query language [Harris and Seaborne 2013].
(2) *Isomorphism-based semantics:* Under this type of semantics, the structure of the query (in some potentially application-dependent sense) should be preserved under the image of the permitted mappings; in more practical terms, certain types of variables are restricted to match distinct constants in the database. Since the precise type of isomorphism – i.e., the precise type of structure preserved – may depend on the application, this leaves us with a variety of different possible isomorphism-based semantics, where we can highlight:
   — *No-repeated-anything semantics:* Only injective mappings are allowed, meaning that no two variables can be bound to the same term in a given match.
   — *No-repeated-node semantics:* The injective restriction only applies to variables that map to nodes (or node ids). In edge-labelled graphs, for example, it is common to only require mappings of node variables to be injective, meaning that multiple variables can still be mapped to the same edge labels. This "no-repeated-node" semantics is often preferred in graph matching applications (see, e.g., [Bunke 2000]) where no nodes in the query graph should be "collapsed" as it would change the structure of the query graph.
   — *No-repeated-edge semantics:* The injective restriction only applies to variables that map to edges: in other words, "edge variables" (variables that map to edge ids in E) must be mapped one-to-one, whereas other types of variables (for nodes, labels, attribute properties and values) need not be injective. This semantics is currently used by the Cypher query language [The Neo4j Team 2016].

Example 3.6. In order to illustrate the differences between these semantics, let G be the property graph of Figure 3 and Q the following basic graph pattern:



From evaluating $Q(G)$, we have the following (unrestricted) results:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
n_2	e_2	directs	n_1	Clint Eastwood	e_3	acts_in	n_3	Anna Levine
n_2	e_3	acts_in	n_3	Anna Levine	e_2	directs	n_1	Clint Eastwood
n_2	e_1	acts_in	n_1	Clint Eastwood	e_3	acts_in	n_3	Anna Levine
n_2	e_3	acts_in	n_3	Anna Levine	e_1	acts_in	n_1	Clint Eastwood
n_2	e_2	directs	n_1	Clint Eastwood	e_1	acts_in	n_1	Clint Eastwood
n_2	e_1	acts_in	n_1	Clint Eastwood	e_2	directs	n_1	Clint Eastwood
n_2	e_1	acts_in	n_1	Clint Eastwood	e_1	acts_in	n_1	Clint Eastwood
n_2	e_2	directs	n_1	Clint Eastwood	e_2	directs	n_1	Clint Eastwood
n_2	e_3	acts_in	n_1	Anna Levine	e_3	acts_in	n_1	Anna Levine

All matches are valid under the homomorphism-based semantics. Only the first two matches would be permitted under the no-repeated-anything semantics since the latter seven matches all map multiple variables to the same term. Only the first four matches would be valid under the no-repeated-node semantics since in the latter five matches, the "node variables" x_4 and x_8 map to the same node. Only the first six matches would be valid under the no-repeated-edge semantics since in the latter three matches, the "edge variables" x_2 and x_6 map to the same edge.

As the example suggests, the appropriate selection of semantics may vary from application to application: no one semantics fits all. □

While some of the previous semantics may restrict the duplication of terms within a single match – namely the isomorphism-based semantics – we can also consider an orthogonal choice of semantics with respect to duplicate matches in the result of evaluating a bgp Q over a graph database G, as follows:

— *Set semantics:* $Q(G)$ is defined as a *set* of matches; in other words, the result of evaluating Q over G cannot contain duplicate matches.
— *Bag semantics:* $Q(G)$ is defined as a *bag* of matches; more specifically, the number of times a match appears in the result corresponds with the number of unique mappings that witness the match.

In fact, on the level of bgps, duplicate matches cannot occur, and hence the set and bag semantics are equivalent. However, when we later extend bgps with features such as projection, union, etc., duplicate matches can occur, distinguishing both semantics.

We can then consider, for example, homomorphism-based set semantics, or isomorphism-based bag semantics, and so forth. Since in much of our discussion it will be inessential which underlying semantics we use for evaluating bgps, we may refer to $Q(G)$ as the evaluation of bgp Q over a graph database G in a generic manner, where we assume a homomorphism-based set semantics unless otherwise stated.

3.2. Complex graph patterns

In terms of traditional relational operations, basic graph patterns (bgps) cover the natural join, and selection based on equality (since constants can be embedded into a bgp). Complex graph patterns (cgps) extend bgps with further traditional relational operations – namely projection, union, difference, optional (aka. left-outer-join) and filter (which covers selection). We will now go through each of these features in turn.

Projection. We call the set of variables for which $Q(G)$ potentially returns matches the *output variables* of the graph pattern Q (which is independent of G). For a bgp, this

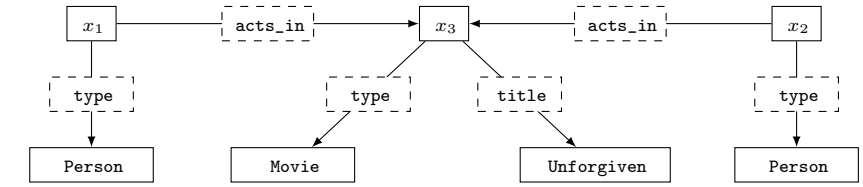
is always the set of all variables in a query. However, projection allows for selecting a subset of the output variables of a graph pattern as the new output variables: it allows for stating which variables are deemed relevant in the evaluation of a cgp. For instance, in Example 3.6, to retrieve only the names of actors who starred together in `Unforgiven` – e.g., for a user who is uninterested in node or edge ids – we can project variables x_5 and x_9; other columns will then be simply omitted from the results. As expected, this operator is present in all practical query languages for graphs, often using the projection keyword `SELECT` as used by SQL.

Join. While the join of two bgps can be easily expressed as another bgp (under homomorphism-based semantics), more complex graph patterns or different semantics require the explicit use of this operator. This corresponds to the usual relational join (more specifically, a *natural join*) over the queries that are defined by two graph patterns Q_1 and Q_2. The output variables of this join corresponds to the union of the output variables of Q_1 and Q_2, and its evaluation contains all matches that can be obtained by joining a match in the evaluation of Q_1 with a match in the evaluation of Q_2. More specifically, two such matches can be joined when they take the same values for the variables that are shared by the output variables of Q_1 and Q_2; in this case, we say that the matches are *compatible*. An explicit join is essential in any query language that goes beyond bgps to combine results from different operations.

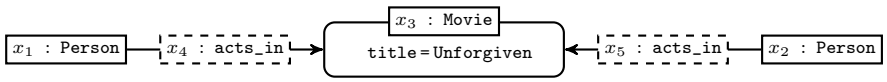
Union and difference. Let Q_1 and Q_2 be two graph patterns. The union of Q_1 and Q_2 is a complex graph pattern whose evaluation is defined as the union of the evaluations of Q_1 and Q_2; for example, in a movie database such as the one from Figure 3, one could use union to find the movies in which Clint Eastwood acted *or* which he directed. The difference of Q_1 and Q_2 is also a complex graph pattern whose evaluation is defined as the set of matches in the evaluation of Q_1 that do not belong to the evaluation of Q_2; for example, one could use difference to find the movies in which Clint Eastwood acted but did not direct. Computing the union of two sets of matches is rather simple in computational terms, and as expected most systems implement the union operator. However, difference is computationally more difficult for certain evaluation problems and as such some systems prefer to leave its implementation out. In some other cases, the implementation of the difference operator has been delayed for future revisions of the language, as was the case for SPARQL, where an explicit difference operator, called `MINUS`, was only introduced in SPARQL 1.1 [Harris and Seaborne 2013].[7]

Optional. This operator is based on the join of two graph patterns Q_1 and Q_2, but instead of dismissing those matches in the evaluation of Q_1 that cannot be joined with a match in the evaluation of Q_2, it keeps them in the result in order to maximise the amount of information retrieved. This feature is particularly useful when dealing with incomplete information, or in cases where the user may not know what information is available. For example, in the context of Figure 5, information relating to the gender of users is incomplete but may still be interesting to the client, where available. Let us assume that the client wishes to retrieve users that follow the U2 tag, where available, to find out what their genders are. Using a natural join, users such as Julie Freud that do not have an explicit gender would be excluded from the results. But instead by using optional, users without a gender will be returned and the value for gender in the corresponding match will simply be left undefined/blank. This operation, then, supports partial answers over incomplete data. In relational terms, the optional operator

[7]We briefly note that SPARQL supports a variant of `UNION` and `MINUS` where, if the output of the base patterns Q_1 and Q_2 differ, then *compatible* matches are unioned or removed, respectively [Pérez et al. 2009; Angles and Gutierrez 2016]. In the case of union, this may create partial matches.



(a) Example for an edge-labelled graph



(b) Example for a property graph

Fig. 7. Two versions of a basic graph pattern to retrieve all pairs of co-stars for the movie Unforgiven: one for a graph database and one for a property graph

corresponds to the *left-outer join* [Galindo-Legaria and Rosenthal 1997]. The optional operator has been present in SPARQL since the original version [Prud'hommeaux and Seaborne 2008; Pérez et al. 2009], and is also included, for example, in the Cypher query language [The Neo4j Team 2016].

Filter. Users may wish to restrict the matches of a cgp over a graph database G based on some of the intermediate values returned using, e.g., inequalities, or other types of expressions. For instance, with respect to Example 3.2, a client may be interested in finding things that mutual friends both liked during *October 2015*, in which case, the client could apply a filter on the cgp of the following form:

$$\texttt{01-10-15} \leq x_5 \leq \texttt{31-10-15} \;\; \text{AND} \;\; \texttt{01-10-15} \leq x_6 \leq \texttt{31-10-15}$$

Applying a filter over a graph pattern does not change its output variables. In general, the filter expression covers the usual conditions permitted by the selection relational operator, including inequalities; boolean connectives such as AND, OR or NOT; etc. However, while basic filter operators are present in some form for all practical graph-based query languages, in certain languages a wide range of expressions is provided to support complex filtering criteria, including regular expressions over strings, arithmetic operators, casting, etc. We give some examples in the following section.

3.3. Graph patterns in practice

We now take a closer look at how graph patterns are applied in three practical query languages: SPARQL, Cypher and Gremlin. We choose these languages because they are the most widely-used query languages in practice but offer significant differences: SPARQL operates over RDF graphs; Cypher is designed to operate over property graphs as defined previously; meanwhile, Gremlin is more imperative in nature than the other two, and is geared more towards graph traversal than graph pattern matching. Given that each of these three languages is associated with lengthy documentation, in the following our goal is not to be complete in discussing the graph pattern matching features of all three engines, but rather to give a quick comparative impression of each language through examples (for which we will use the bgps depicted in Figure 7) and to highlight and contrast some important aspects.

SPARQL. SPARQL is a declarative language recommended by the W3C for querying RDF graphs [Prud'hommeaux and Seaborne 2008; Harris and Seaborne 2013]. The basic building blocks of SPARQL queries are *triple patterns*, which are RDF triples

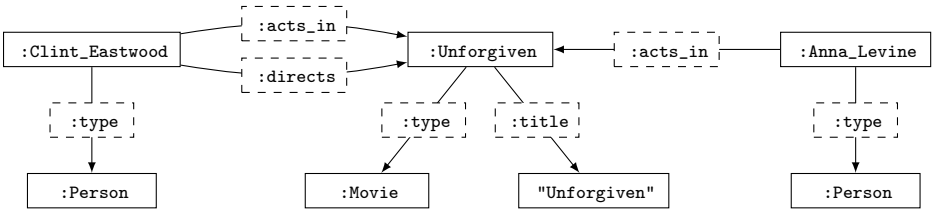


Fig. 8. RDF graph extending the edge-labelled graph of Figure 1

where the subject, object or predicate may be a variable (variables in SPARQL typically start with the symbol '?'). Several triple patterns can be combined (conjunctively) into a basic graph pattern. On top of basic graph patterns, SPARQL also supports all of the complex graph pattern features discussed previously (and more besides). The evaluation of bgps in SPARQL is done following homomorphism-based bag semantics. In the following, we will use Figure 8 as our example RDF data.

Example 3.7. The following SPARQL query represents a complex graph pattern that combines the basic graph pattern of Figure 7(a) with a projection that asks to only return the co-stars and not the movie identifier:

```
PREFIX : <http://ex.org/#>
SELECT ?x1 ?x2
WHERE {
  ?x1 :acts_in ?x3 . ?x1 :type :Person .
  ?x2 :acts_in ?x3 . ?x2 :type :Person .
  ?x3 :title "Unforgiven" . ?x3 :type :Movie .
  FILTER(?x1 != ?x2)
}
```

Recalling that constants in RDF graphs can be IRIs, the purpose of the `PREFIX` statement is to define a shortcut for a namespace under which constants appear; since prefixes are inessential to our discussion, we will henceforth leave them implicit. In the `SELECT` clause, we specify the variables we wish to project as output. The `WHERE` clause then captures the basic graph pattern of Figure 7(a): it contains six triple patterns (delimited by periods) that correspond to the edges of Figure 7(a). Additionally, since the semantics of SPARQL evaluation is homomorphism-based, we add a `FILTER` to ensure that we do not match cases where `?x1` and `?x2` map to the same person.

Applied to Figure 8, this query would thus return:

?x1	?x2
:Clint_Eastwood	:Anna_Levine
:Anna_Levine	:Clint_Eastwood

Other matches for the bgp are removed by the filter and `?x3` is projected away. □

The previous example shows how bgps, projection and filter are supported in SPARQL. We now look at some brief examples for the remaining cgp features that are all based on the graph database of Figure 8.

Example 3.8. We start with an example of a union to find movies that Clint Eastwood has acted *or* directed in.

```
SELECT ?x
WHERE {{ :Clint_Eastwood :acts_in ?x . } UNION { :Clint_Eastwood :directs ?x . }}
```

Both patterns to the left and right of the UNION will be evaluated *independently* and their results unioned. This will return :Unforgiven; in fact, this result will be returned twice since SPARQL, by default, adopts a bag semantics. □

Example 3.9. We could use difference to ask for people who acted in the movie Unforgiven but who did not (also) direct it:

```
SELECT ?x
WHERE {{ ?x :acts_in :Unforgiven . } MINUS { ?x :directs :Unforgiven . }}
```

Any match for the left side of the MINUS that is compatible with a match from the right side will be removed. Hence, this query will return :Anna_Levine. □

Example 3.10. Using optional, we could ask for movies that actors have appeared in, and any other participation they had with the movie besides acting in it:

```
SELECT ?x1 ?x2 ?x3
WHERE {{ ?x1 :acts_in ?x2 .} OPTIONAL { ?x1 ?x3 ?x2 . FILTER(?x3 != :acts_in) }}
```

This will return:

?x1	?x2	?x3
:Clint_Eastwood	:Unforgiven	:directs
:Anna_Levine	:Unforgiven	

A result is still returned for :Anna_Levine even though she had no other participation in the movie; instead the relevant column is left blank for that result. □

In the latter example, we show how optional and filter can be combined. Of course, it is also possible to combine these features in other ways to form increasingly more complex graph patterns, for example, to find movies Clint Eastwood has neither acted nor directed in, or to find his co-stars in those movies he did not also direct, etc.

Here we have provided a few brief examples of the most notable features for graph patterns that SPARQL supports. However, the list of graph pattern features we cover is far from complete, where for example SPARQL 1.1 now supports a wide range of FILTER expressions, variable assignments, arithmetic operations, conditionals, federation, and so forth. Likewise, rather than operate over a single graph, SPARQL operates over collections of graphs, called "Named Graphs", which allow for selecting customised partitions of the data over which queries should be executed. We refer to the official standard for more details [Harris and Seaborne 2013]. Other SPARQL features such as property paths will be covered in later sections.

Cypher. Cypher is a declarative language for querying property graphs that uses "patterns" as its main building blocks [The Neo4j Team 2016]. Patterns are expressed syntactically following a "pictorial" intuition to encode nodes and edges with arrows between them. Unlike SPARQL, Cypher uses isomorphism-based no-repeated-edges bag semantics. We again give a quick flavour of Cypher in some examples, where this time we will consider evaluation against the property graph of Figure 3.

Example 3.11. The pattern in Figure 7(b) would be written in Cypher as:

```
MATCH (x1:Person) -[:acts_in]-> (:Movie {title:"Unforgiven"})
                  <-[:acts_in]- (x2:Person)
RETURN x1,x2
```

The MATCH clause specifies the bgp in question. Nodes are written inside "( )" brackets and edges inside "[ ]" brackets. Filters for labels can be written after the node separated with a ":" symbol, such that (x1:Person) represents a node x1 that must match

to a node labelled Person. Specific values for properties can be specified within "{ }" brackets; for instance (:Movie {title:"Unforgiven"}) represents a node that must match to a node labelled Movie and that must have value Unforgiven for the property title. The RETURN clause can be used to project the output variables. Implicit projection is also allowed inside the pattern itself by simply omitting some of the variables; we have done this for the edges and the node with label Movie.

Cypher implements a no-repeated-edge semantics, and thus the evaluation of this query against the movie graph of Figure 3 would not include the match that sends both x1 and x2 to the node of Clint Eastwood (that is, n1) since it would require mapping to the same edge e_1 twice in a single match (and likewise for the match that sends x1 and x2 to the node of Anna Levine). One possibility to overcome this restriction is to use the explicit (natural) join operation of Cypher, which is invoked by simply including additional MATCH commands. For example, if we wanted to construct a pattern that retrieves all pairs of actors who act in the same movie, including pairs that repeat the same actor, we would use the following Cypher statement:

```
MATCH (x1:Person) -[:acts_in]-> (x3:Movie {title:"Unforgiven"})
MATCH (x2:Person) -[:acts_in]-> (x3)
RETURN x1,x2
```

This is equivalent to the natural join of the evaluations of the two patterns given by the two MATCH statements. In this case, we would also get the matches that send both x1 and x2 to the node of Clint Eastwood (and likewise to the node of Anna Levine). □

If a variable x stores a node or edge id, Cypher offers an "." operator to refer to the value of some property of x. For instance, in our previous example we can refer to the value of the property name for the variable x1 by using notation "x1.name", and thus use "RETURN x1.name,x2.name" to return the actors' names (rather than their node ids).

Cypher supports union, difference, optional, and filter. We now provide similar example queries as for SPARQL, this time against the property graph of Figure 3.

Example 3.12. In the following query, we use union to ask for the titles of movies that Clint Eastwood either acted in or directed:

```
MATCH (:Person {name:"Clint Eastwood"}) -[:acts_in]-> (x3:Movie)
RETURN x3.title
UNION ALL MATCH (:Person {name:"Clint Eastwood"}) -[:directs]-> (x3:Movie)
RETURN x3.title
```

Both patterns will be evaluated independently and their results unioned. The "ALL" keyword indicates that duplicates should be returned; in this case, the title Unforgiven will be returned twice. Omitting the "ALL" keyword, the title would appear once. □

Example 3.13. We can use difference to return the people who acted in but did not direct the movie Unforgiven:

```
MATCH (x1:Person) -[:acts_in]-> (x3:Movie {title:"Unforgiven"})
WHERE NOT  (x1) -[:directs]-> (x3)
RETURN x1.name
```

The "NOT" keyword indicates the difference operator: any match for the initial pattern that is compatible with a match for the pattern indicated after "NOT" will be removed. In this case, Anna Levine will be returned. □

Example 3.14. We now use optional and filter to find movies in which people have acted and other ways they participated in the movie, if any.

```
MATCH (x1:Person) -[:acts_in]-> (x3:Movie)
OPTIONAL MATCH (x1) -[x4]-> (x3)
WHERE type(x4) <> "acts_in"
RETURN x1.name AS name, x3.title AS movie, type(x4) as part
```

In this query, the `WHERE` clause is a true filter expression: "<>" denotes inequality and "`type`" is a built-in function to return the label of an edge. The first match will retrieve all pairs of actors and movies, where the second optional match will check the other edges between each such pair matching edges where the label is not `acts_in`.

Cypher allows the use of the operator `AS` in the `RETURN` clause to indicate that the results of the query should be displayed under some specific names for columns. For instance, the use of "`x3.title AS movie`" indicates that the values of the property `title` of the nodes stored in the variable `x3` will be displayed in a column with name `movie`. Hence, the query in this example returns:

name	movie	part
Clint Eastwood	Unforgiven	directs
Anna Levine	Unforgiven	

Given that we use the optional matching functionality, we see that the result for the `Anna Levine` node is preserved even though she only acted in the movie. □

Once again, here we only provide examples of the *core* matching features supported by Cypher to give a flavour of the language; we refer the interested reader to the online documentation for further details [The Neo4j Team 2016].

Gremlin. The last language we review is Gremlin: the query language of the Apache TinkerPop3 graph Framework [Apache TinkerPop 2017]. Although Gremlin is also specified with the property graph model in mind, it differs quite significantly from the previous two declarative languages and has a more "functional" feel: while SPARQL and Cypher have obvious influences from SQL for example, Gremlin feels more like a programming language interface.[8] Likewise, its focus is on navigational queries rather than matching patterns; however, amongst the "graph traversal" operations that it defines, we can find familiar graph pattern matching features. Similarly to SPARQL, Gremlin also uses the homomorphism-based bag semantics.

Example 3.15. Intuitively, Gremlin traversals give explicit instructions as to how the graph is to be navigated. For example, to retrieve all movies where `Clint Eastwood` is an actor, we first load a "graph traversal" object (labelled `G` here) and write:

```
G.V().hasLabel('Person').has('name','Clint Eastwood')
  .out('acts_in').hasLabel('Movie')
```

The call `G.V()` will return the set of all nodes in the graph (`V` stands for "vertex"). We then apply two selections on the set of nodes, where the sequence of calls `G.V().hasLabel('Person').has('name','Clint Eastwood')` retrieves precisely those nodes with label `Person` and name `Clint Eastwood`. The command `out('acts_in')` retrieves all nodes that can be reached from these latter nodes with an edge labelled `acts_in`. Finally `hasLabel('Movie')` filters nodes not labelled with `Movie`. □

Gremlin is most natural when expressing paths because all such patterns can be simulated by a traversal on the graph.

[8]Strictly speaking, Gremlin is a functional language that includes several operators that are out of the scope of this survey. We concentrate on querying functionalities, denoted as "graph traversals" in the documentation [Apache TinkerPop 2017]. There are various versions of Gremlin for integration with different programming languages. Here we stick with Gremlin-Groovy.

Example 3.16. The following Gremlin traversal allows us to obtain all co-actors of `Clint Eastwood`:

```
G.V().hasLabel('Person').has('name','Clint Eastwood')
  .out('acts_in').hasLabel('Movies')
    .in('acts_in').hasLabel('Person')
```

This query navigates through the movies of `Clint Eastwood` as before, but then continues: the command `in('acts_in')` looks for nodes that are connected by an edge labelled `acts_in` in the opposite direction as the traversal, and then `hasLabel('Person')` again filters out any nodes that are not of label `Person`. □

Nevertheless, Gremlin does include a way of specifying more general bgps (including branches and cycles): traversals are used to encode the structure, but nodes can be cross-referenced at different points using variables specified by means of the `as` command, while the pattern is then evaluated using the `match` command.

Example 3.17. To illustrate a more complex example, we show how the bgp in Figure 7(b) can be expressed in Gremlin. The following example additionally includes an explicit filter to ensure that `x1` does not map to the same constant as `x2` in any match, and also adds a projection to return only results for the `x1` and `x2` variables (in this case returning only the co-stars, not the movie they co-starred in).

```
G.V().match(
  __.as('x1').hasLabel('Person').out('acts_in').hasLabel('Movies').as('x3'),
  __.as('x3').has('title','Unforgiven').in('acts_in').hasLabel('Person').as('x2'),
  .where('x1', neq('x2'))
).select('x1','x2')
```

Again `G.V()` returns all vertices in the graph. The `match` command then takes a list of arguments; in this case, the command takes three arguments that specify two inner traversals and a filter. The '`__`' operator means that the subsequent operation is applied on the parent traversal one level up, meaning that, for example, "`__as('x1')`" will apply over all nodes in `G.V()`. The '`as`' command declares a variable; however, rather than "`__as('x1')`" binding all nodes to variable `x1`, the entire traversal acts as a bgp, meaning that subsequent steps from a node in `x1` must be satified for the variable to match that node. Each inner traversal can thus be seen as a tree-shaped bgp. These inner traversals are then joined to create a more complex bgp that may contain cycles. In the above example, the two inner traversals are accompanied by a `where` command that calls a not-equals (`neq`) filter to ensure that `x1` and `x2` are not bound to the same result. The `select` command (outside `match`) then performs a projection to select the output of the query: only the co-stars, not the movie. □

While Gremlin supports bgps, filters and projection, its main focus is on navigational queries, which will be discussed in Section 4. The current version has limited support for declarative-style operators for complex graph patterns. While a "`union`" command exists, and difference can be emulated by the "`drop`" command, the current version does not have explicit support for optional. We will not go into details but instead refer the interested reader to the online documentation [Apache TinkerPop 2017].

3.4. The complexity of evaluating graph patterns

To understand the computational complexity of working with a query language we consider the following evaluation problem: given a query Q in this language, a possible answer h and a graph database or property graph G, verify whether h is an answer to Q over G; that is, verify whether $h \in Q(G)$. The most basic fragment of graph query languages that needs to be considered is the fragment consisting of bgps and

projection, which corresponds to conjunctive queries in relational databases [Abiteboul et al. 1995]. The evaluation problem for this fragment is NP-complete for the homomorphism-based semantics and the three versions of the isomorphism-based semantics considered in Section 3.1; NP-hardness can be proven for the former by reduction from the graph homomorphism problem [Hell and Nesetril 2004], while for the latter, it can be established by reduction from the subgraph isomorphism problem [Ullmann 1976]. All of these results hold under set or bag semantics since the question of "$h \in Q(G)$?" is not affected by such a choice of semantics.

In practice the size of the query Q is typically much smaller than the size of the database G, so it is common practice to assign different roles to the two when analysing query evaluation. This motivated the introduction of the notion of *data complexity* [Vardi 1982], in which Q is assumed to be fixed and the input is given by G only; this is in contrast to the more general notion of *combined complexity*, which is defined with respect to the input query and the database (as in the previous paragraph). Under data complexity, evaluation of queries consisting of bgps and projection can not only be solved in polynomial time, but also can be carried out in logarithmic space for all the semantics considered in Section 3.1 [Abiteboul et al. 1995]. Although data complexity might seem a bit simplistic at first sight, it has proven useful for understanding the cost of evaluating small queries over datasets of moderate size.

Furthermore, in practice one is often interested in matching simple bgps that are not necessarily that difficult to evaluate. Both the graph theory and the database communities have dedicated vast amounts of work to identifying classes of patterns for which the matching problem can be efficiently solved, even in combined complexity. One of the main results here indicates that, intuitively speaking, the more cyclical the underlying structure of the graph pattern (i.e., the less it resembles a tree), the more difficult the query is to evaluate; this notion of cyclicity is captured formally by a notion call *treewidth*, where we refer the reader to, e.g., [Chekuri and Rajaraman 2000; Dalmau et al. 2002; Gottlob et al. 2016] for detailed discussion.

Going beyond the fragment covering bgps and projection, the combined complexity of the evaluation of cgps has been extensively studied for SPARQL. To recap the main results, let us first consider SPARQL under set semantics. If only projection, join, union and filter are allowed in the language, then the combined complexity of the evaluation problem remains NP-complete. If difference and optional are also allowed, then SPARQL has the same operators as relational algebra, so the combined complexity is PSPACE-complete [Vardi 1982]. Interestingly, it can be proven that the `MINUS` operator of SPARQL can be simulated using optional, filter and join [Angles and Gutierrez 2008], so the complexity of the evaluation problem remains PSPACE-complete without `MINUS`. Moreover, the same complexity bound can be obtained if only join and optional are allowed [Schmidt et al. 2010], but in this case the proof is not based on an expressiveness argument. For the case of SPARQL under bag semantics, the combined complexity of the evaluation problem remains PSPACE-complete. To the best of our knowledge, the complexity of cgps has not been studied for the cases of Cypher and Gremlin, thus opening interesting opportunities for future investigation. We further discuss open questions regarding the complexity of Cypher and Gremlin in Section 5.

4. NAVIGATIONAL QUERIES

While graph patterns allow for querying graph databases in a bounded manner, it is often useful to provide more flexible querying mechanisms that allow to *navigate* the topology of the data. One example of such a query is to find all friends-of-a-friend of some person in a social network such as the one in Figure 5. Here we are not only

interested in immediate acquaintances of a person, but also the people she might know through other people; namely, her friends-of-a-friend, their friends, and so on.

Queries such as the one above are called *path queries*, since they require us to navigate through the graph using paths of potentially arbitrary length. Path queries have long been established as the core of navigational querying in graphs by the research community [Wood 2012; Barceló 2013] and are widely supported in graph database engines [The Neo4j Team 2016; Harris and Seaborne 2013; Apache TinkerPop 2017]. Furthermore, path queries have found applications in areas such as the Semantic Web [Alkhateeb et al. 2009; Pérez et al. 2010; Paths 2009], provenance [Holland et al. 2008] and route-finding applications [Barrett et al. 2000], amongst others. It is therefore natural to add path queries to basic graph patterns as the core of graph querying. We call such queries *navigational queries*, and in this section we discuss how they can be used to query graph databases. We start with path queries.

4.1. Path Queries

Paths are the most basic navigational object in a graph database. The most fundamental type of path query is that of *path existence*, which asks if there is some directed path between two nodes in a property graph, irrespective of edge labels; in some cases, one or all such paths can be additionally returned. This is a foundational notion related to the problems of reachability and transitive closure in directed graphs [Yu and Cheng 2010], and for this reason it has been well studied by the theoretical community. However, in practice, one often needs *path queries* that impose additional constraints on the path that is to be computed, such as restrictions on edge labels. The transitive friend-of-a-friend relation in social networks is such an example: we are interested in paths composed only of edges labelled with `knows` (and not `likes` or any other label).

Definition. We can define a *path query* as having the general form $P = x \xrightarrow{\alpha} y$, where α specifies conditions on the paths we wish to retrieve and x and y denote the endpoints of the path. The endpoints x and y can be variables, or specific nodes, or a mix of both, or even the same node (in which case we are specifying a cycle). For the expression α, we can use the symbol $*$ to signify that we are only interested in the existence of a path connecting two nodes without imposing any further constraints; otherwise, there are a variety of formalisms under which α can express more complex *path constraints* [Cruz et al. 1987; Mendelzon and Wood 1989; Barceló et al. 2012a; Calvanese et al. 2003; Libkin et al. 2016], but probably the most famous is that of *regular expressions* [Hopcroft et al. 2003] defined over the set *Lab* of edge labels. When used as a path constraint, a regular expression specifies all paths whose edge labels, when concatenated, form a word in the language of the regular expression. Intuitively speaking, regular expressions allow for concatenating paths, for applying a union/disjunction of paths, and for applying a path zero or many times. Path queries specified using regular expressions are commonly known as *Regular Path Queries* (RPQs).

Example 4.1. The (transitive) friend-of-a-friend relationship in our social network can be expressed via the following regular path query (RPQ):

$$P := x \xrightarrow{\texttt{knows}^+} y.$$

Here the symbol '+' denotes "one-or-more", where the regular expression $\texttt{knows}^+$ is used to specify all paths formed from a sequence of one-or-more forward-directed edges with the label `knows`.[9] Thus the endpoints x and y would be matched to any two nodes in

[9]Note that $\texttt{knows}^+$ is equivalent to $\texttt{knows} \cdot \texttt{knows}^*$, where '$*$' denotes the Kleene star (zero-or-more) and '$\cdot$' denotes concatenation.

the social network connected by such a path. Similarly, we can use the path query:

$$P' \;:=\; x \xrightarrow{\texttt{knows}^+ \cdot \texttt{likes}} y,$$

where '$\cdot$' denotes concatenation, to match nodes x and y such that x is a person and y is a post that is liked by a (transitive) friend-of-a-friend of x. Finally we can apply a union of paths to match the liked or disliked posts of transitive friends-of-a-friend of x:

$$P'' \;:=\; x \xrightarrow{\texttt{knows}^+ \cdot (\texttt{likes} \,|\, \texttt{dislikes})} y,$$

where the '|' symbol here denotes a union. □

The features of RPQs can be combined to (implicitly) support a number of other navigational operations on graphs. For instance, the RPQ $P = x \xrightarrow{\alpha} y$, with

$$\alpha \;=\; \texttt{knows} \;\big|\; (\texttt{knows} \cdot \texttt{knows}) \;\big|\; \ldots \;\big|\; \underbrace{(\texttt{knows} \cdot \texttt{knows} \cdot \ldots \cdot \texttt{knows})}_{k \text{ times}}$$

defines the friend-of-a-friend relationship up to depth $k \geq 2$. Likewise, for example, the RPQ $x \xrightarrow{Lab^*} y$, where Lab^* is the regular expression that accepts all words over Lab, corresponds to the path query that imposes no constraints on paths. Regardless, we will keep using $x \xrightarrow{*} y$ to express this query, even when talking about RPQs.

However, there are various navigational operations not supported by RPQs that seem quite natural. RPQs are sometimes thus extended to allow further expressions. One such extension is to allow an *inverse* operator a^- (for a in Lab) to specify the traversal of edges in a backwards direction, giving rise to *Two-way Regular Path Queries* (2RPQs), which are RPQs enhanced with inverses [Calvanese et al. 2002; Calvanese et al. 2003].

Example 4.2. Consider now a movie database such as the one in Figure 3. The following two-way regular path query (2RPQ) retrieves all co-stars in the database:

$$P \;:=\; x \xrightarrow{\texttt{acts\_in} \cdot \texttt{acts\_in}^-} y.$$

The expression `acts_in` matches a node x against a person, then the path navigates to the movies that x starred in, and then backwards to x's co-stars (or to x itself). Similarly, we can use the path query:

$$P' \;:=\; x \xrightarrow{(\texttt{acts\_in} \cdot \texttt{acts\_in}^-)^+} y$$

to compute the transitive closure of the co-star relationship; for example, if we wished to check which actors have a finite *Bacon number* [Reynolds 2015] – i.e., which actors have transitively co-starred in a movie with the actor Kevin Bacon – we could use this pattern, setting x to `Kevin Bacon` and leaving y as a variable. □

The need for RPQs (and their extended forms) has been long argued by the research community [Buneman et al. 1996; Buneman 1997] and recently they have been implemented in various systems; for example, extensions of RPQs form the conceptual core of "property paths" in the SPARQL 1.1 standard [Harris and Seaborne 2013], which have been implemented in the newest versions of various SPARQL engines [Bishop et al. 2011; Erling 2012; Thompson et al. 2014] and have been studied by numerous authors [Arenas et al. 2012; Losemann and Martens 2013; Fionda et al. 2015; Kostylev et al. 2015b]. Likewise in the Cypher query language [The Neo4j Team 2016], one can find RPQ-like features. We will provide examples of the use of RPQ-like features in such languages later in this section.

Evaluation. To define how path queries are evaluated we need to formalise the notion of a path over graph databases. In a property graph G, a path π is a sequence $n_1e_1n_2e_2n_3 \dots n_{k-1}e_{k-1}n_k$, where $k \geq 1$ and with each e_i being an edge in G between n_i and n_{i+1}. The label of the path π, denoted $\mathit{Lab}(\pi)$, is the concatenation of its edge labels, namely $\mathit{Lab}(\pi) = a_1a_2 \dots a_{k-1}$, where a_i is the label of e_i. For example, the sequence $n_1e_1n_2e_6n_5$ is a path in the property graph of Figure 5. The label of the path is the word $\texttt{knows} \cdot \texttt{dislikes}$. Note that for each node n of G the sequence that consists exclusively of n is also a path (of length zero). The label of such zero-length paths corresponds to the empty word, denoted by ϵ.

To define paths in edge-labelled graphs we need to be more careful since we do not have edge identifiers in this model, and thus we cannot give the same definition as before. Instead, we define a path π in an edge-labelled graph G as a sequence: $n_1a_1n_2a_2n_3 \dots n_{k-1}a_{k-1}n_k$, where (n_i, a_i, n_{i+1}) is an edge in G for all $i < k$. In this case the label is simply $\mathit{Lab}(\pi) = a_1a_2 \dots a_{k-1}$. As in the case of property graphs, a single node n forms a zero-length path with the label ε.

The *evaluation of a path query* $P = x \xrightarrow{\alpha} y$ over G, denoted $P(G)$, then consists of all paths in G whose label satisfies α. For instance, if $\alpha = *$, any path belongs to $P(G)$, but if α is the regular expression L, then only paths whose label belongs to L appear in $P(G)$.[10] The set of paths matching $P(G)$ might be infinite (when G has directed cycles), and thus this general definition of evaluation is not computable. Later we will see different ways in which this definition is restricted to be implemented in practice.

Example 4.3. Let G denote the property graph of Figure 5 and consider the RPQ $P = x \xrightarrow{\texttt{knows}^+} y$. Because of the cycle between nodes n_1 and n_2 in G, the number of paths in $P(G)$ is infinite: it contains all finite sequences of the form $n_1e_1n_2e_2n_1e_1 \cdots$ and $n_2e_2n_1e_1n_2e_2 \cdots$. For the case of the RPQ $P' = x \xrightarrow{\texttt{knows}^+ \cdot \texttt{likes} \cdot \texttt{hasTag}} y$, the following table shows a few paths in $P'(G)$:

n_1	e_1	n_2	e_7	n_4	e_4	n_3						
n_1	e_1	n_2	e_2	n_1	e_5	n_4	e_4	n_3				
n_1	e_1	n_2	e_2	n_1	e_1	n_2	e_7	n_4	e_4	n_3		
n_1	e_1	n_2	e_2	n_1	e_1	n_2	e_2	n_1	e_5	n_4	e_4	n_3
$\vdots$						$\vdots$						$\vdots$

The number of paths in $P'(G)$ is also infinite. □

As in the case of graph patterns, different practical considerations – for example, the possibility of having paths involving cycles – give rise to different semantics for the evaluation for path queries, or more specifically, for which paths are included in $P(G)$. Next we describe the most common such forms of evaluation in practice:

(1) *Arbitrary path semantics:* All paths are considered. More specifically, all paths in G that satisfy the constraints of P are included in $P(G)$. As per Example 4.3, under this semantics, $P(G)$ may contain an infinite number of paths. However, while it may not be feasible to enumerate all paths under this semantics, a user may only be interested in whether or not such a path exists, or in the (finite) pairs of nodes connected by such paths, etc., in which case such a semantics can be practical [Calvanese et al. 2003; Wood 2012; Barceló et al. 2012a].

[10] From a formal point of view we can treat 2RPQs (path queries with inverses) as standard RPQs that are evaluated over the *completion* of G, which is constructed by adding an edge labelled a^- from v to u for each edge labelled a from u to v. Hence from now on we will consider RPQs to always contain inverses.

(2) *Shortest path semantics:* In this case, $P(G)$ is defined in terms of *shortest paths* only, i.e., paths of minimal length that satisfy the constraint specified by P. We may use this semantics when we want to find pairs of nodes that are linked by some path and, for each such pair, a minimal path (or set of minimal paths of equal length) that witness(es) this. In Example 4.3, the shortest path for $P'(G)$ corresponds to the first path in the table.
(3) *No-repeated-node semantics:* In this case, $P(G)$ contains all matching paths where each node appears once in the path; such paths are commonly known as *simple paths*. This interpretation makes sense in some practical scenarios; e.g., when finding a route of travel, it is often not desired to have routes that come to the same place more than once. The interaction of this interpretation with RPQs has been studied in depth by the theoretical community [Mendelzon and Wood 1989; Arenas et al. 2012; Losemann and Martens 2013]. In Example 4.3, only the first path for $P'(G)$ would be selected since others mention a node more than once.
(4) *No-repeated-edge semantics:* Under this semantics, $P(G)$ contains all matching paths where each edge appears only once in the path. The Cypher query language of the Neo4j engine currently uses this semantics (see Section 3.4.1. of the Cypher Manual [The Neo4j Team 2016]). Use-cases for this semantics are similar as for the previous one; e.g., when we want to visit some place more than once, but we do not want to take the same route as before. In Example 4.3, the first two paths in $P'(G)$ have no repeated edge, but the other paths would not be considered.

Output. As hinted at previously, a user may have different types of questions with respect to the paths contained in the evaluation $P(G)$, such as: Does there exist any such path? Is a particular path π contained in $P(G)$? What are the pairs of nodes connected by a path in $P(G)$? What are (some of) the paths in $P(G)$? We can categorise such questions by what they return as results:

— *Boolean:* In some cases, the output of a path query may be a true/false value to ascertain, for example, if $P(G)$ is non-empty, or if there exists a path in $P(G)$ between two particular nodes, etc.
— *Nodes:* In some applications, we are interested in the nodes connected by specific paths (see, e.g., [Wood 2012; Barceló 2013]). In such cases, we project from $P(G)$ the endpoint nodes: all pairs of nodes u and v linked by some path in $P(G)$. Referring back to Example 4.3, we would project from $P'(G)$ the node pair (n_1, n_3).
— *Paths:* In this case, some or all of the full paths are returned from $P(G)$. For example, if $P(G)$ is applied with a shortest-path semantics, then we would return one or more such shortest paths. In other cases, paths to be returned may be selected based on more complex conditions, e.g., based on a ranking on paths; this may be useful in, e.g., route finding applications, where some top-k "best" paths are sought.
— *Graphs:* Another solution – for example under arbitrary path semantics – is to offer a *compact* representation of the output, e.g., in the form of another graph whose paths are precisely the paths in the output of the query [Barceló et al. 2012a].

While the first two types of answers can be handled under, e.g., a standard relational algebra, there is currently no consensus on how to represent paths as the output of a query. In particular, unlike solutions to graph patterns that have a fixed-arity output, paths do not have a fixed-arity, therefore we cannot directly define a mapping from variables to constants as in the case of a bgp match. Likewise, although returning graphs as queries is supported in SPARQL [Harris and Seaborne 2013] through `CONSTRUCT`, graph creation is only supported as a final step, where such graphs cannot be manipulated further by other operators.

Sets vs. bags. In the case of queries that return a boolean value or a graph as a result, there is no distinction between bag or set semantics. Likewise, in the case that full paths – i.e., the complete sequence of nodes and edges in each path – are returned, no duplicates can occur and there is no such distinction. However, if nodes are returned, or nodes/edges are projected from a full path, then bag semantics are distinguished from set semantics. In particular, if we consider the case where we are returning end nodes of our path as output, when using set semantics, a pair (n, n') will be returned exactly once when there is at least one path in $P(G)$ connecting n with n', and zero times otherwise; when using bag semantics, this now changes, and a pair (n, n') is returned once for each full path in $P(G)$ connecting n with n'.

Bag semantics combined with arbitrary path semantics is problematic since the set of paths can be infinite; thus this combination is usually not considered in the theoretical literature [Wood 2012; Barceló 2013]. But even when the number of paths is guaranteed to be finite, there are still several issues with respect to high computational complexity since bag semantics implicitly requires counting paths. For example, it is well-known that counting the number of paths without repeated nodes from node a to node b in a graph G is a #P-complete problem [Valiant 1979], which implies that it is as difficult as, for example, counting the number of satisfying assignments of a propositional formula, or counting the number of Hamiltonian cycles in a graph.

This high computational complexity has a number of practical consequences. For instance, the initial combination of bag semantics with property paths in drafts of the SPARQL 1.1 standard required that the number of repetitions of a pair of nodes in the answer was equal to the number of paths between them. Thus, a restriction to consider simple paths was added to guarantee finiteness of results. Unfortunately, this gave rise to a path counting problem with a very high complexity [Arenas et al. 2012; Losemann and Martens 2013], which was resolved by imposing a set semantics on property paths of the form `(p)*` and `(p)+`, avoiding the counting of paths of unbounded length. On the other hand, Cypher maintains a bag semantics when returning nodes, where a no-repeated-edge semantics is applied by default.

4.2. Adding paths to basic graph patterns

Now that we understand how path queries can be used to match paths and how graph patterns can be used to match sub-graphs, we can combine them to produce a powerful query language that allows to find more flexible matches. In particular, this language allows to express that some edges in a graph pattern should be replaced by a path (satisfying certain conditions) instead of a single edge.

Example 4.4. In Example 4.2, we used the query $Q' = x \xrightarrow{(\texttt{acts\_in}\cdot\texttt{acts\_in}^-)^+} y$ to find actors that are connected through co-star relations to other actors, and mentioned that this query can be used to find actors with a finite Bacon number. To make our example more challenging, consider now that our movie database from Figure 3 is extended to also contain bibliographical information about scientific papers and their authors. In such a database, each node is either a movie, a person, or an article. Persons and movies are connected as in Figure 3, while a person can also have an `author` edge connecting it to an article. In such a database we might be interested in finding people with finite Erdős–Bacon number, that is, people who are connected to Kevin Bacon through co-stars relations and are connected to Paul Erdős through co-authorship relations. This is easily expressed using the query in Figure 9, which is a basic graph pattern that permits (two-way) regular path queries on edges. □

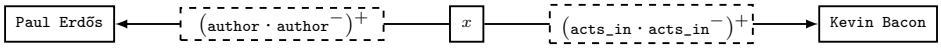


Fig. 9. A query finding the actors with a finite Erdős–Bacon number over an edge-labelled graph

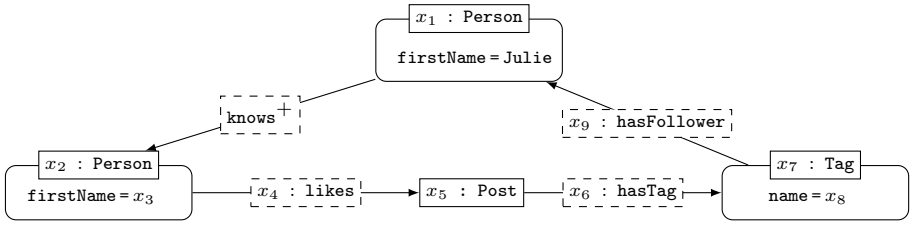


Fig. 10. A navigational graph pattern that characterises the friends of friends of Julie that like a post with a tag she that she follows

Combining path queries with basic graph patterns (bgps) gives rise to *navigational graph patterns* (ngps). In the case of edge-labelled graphs, ngps are defined similarly as bgps: namely, they are edge-labelled graphs where nodes can be constants or variables, and the edge labels can be constants, variables, RPQs[11], or the special symbol $*$ denoting an arbitrary path. Matches are defined as in the case of bgps, but now every edge *not* labelled with a variable is mapped to a path. That is, if we have (b, α, c) in our ngp, with α either $*$ or a regular expression, then our match h must satisfy that $h(b)$ is connected to $h(c)$ by a path in $P(G)$, with P being the path query $x \xrightarrow{\alpha} y$. Note that in order to keep the arity of matchings bounded by the size of the query, we are opting for an *existential* interpretation of path expressions in ngps. That is, we are considering the boolean output semantics for P, which only checks that there is a path in $P(G)$ connecting the nodes $h(b)$ and $h(c)$, but does not return such a path. Navigational graph patterns for property graphs are defined analogously, but now allowing for elements of property graphs in nodes and edges as per Definition 2.3. In particular, if the label α of the edge is $*$ or a regular expression, the end nodes of this edge have to be in the answer to the path query $x \xrightarrow{\alpha} y$ over G.

Example 4.5. Coming back to the social network from Figure 5, we might be interested in finding all friends of friends of Julie that liked a post with a tag that Julie follows. The navigational graph pattern in Figure 10 expresses this query over the property graph of Figure 5. □

Navigational graph patterns have received a lot of attention in the theoretical literature under the name *conjunctive regular path queries* (CRPQs) [Consens and Mendelzon 1990; Florescu et al. 1998; Calvanese et al. 2003; Barceló et al. 2014]. A natural extension of ngps is to consider *complex navigational graph patterns* (cngps) by taking the closure of ngps under the relational operations of selection, projection, join, union, difference, and optional, as presented in Section 3. Some other variants and extensions of cngps allowing to compare different paths in a graph have also been considered in the past [Barceló et al. 2012a; Barceló and Muñoz 2014; Figueira and Libkin 2015]. As we will see later in Section 4.3, cngps then form the core of languages such as SPARQL.

Example 4.6. To give a brief idea of the expressivity of cngps, consider the ngp of Example 4.5 and assume we project x_5: the ids of the posts liked by friends-of-friends of Julie and that have a tag that she follows. Let's call these results the "recommended

[11] In the context of ngps we identify the expression defining an RPQ with the RPQ itself.

posts" for Julie. Now consider a copy of the same pattern to find the recommended posts for John. We could use the union of these patterns to find posts recommended for Julie or John, or intersection to find posts recommended for both, or difference to find posts recommended for Julie but not John, or filter dates to find more recent posts, and so forth. All such queries can then be expressed as cnpgs. □

For the navigational languages we have seen thus far, paths are the only form of recursion allowed. However, to express certain types of queries, we may require more expressive forms of recursion. In the online appendix we present three families of such languages, namely, *nested regular expressions* [Pérez et al. 2010; Barceló et al. 2012b], *regular data path queries* [Libkin and Vrgoč 2012], and *Datalog extensions* [Consens and Mendelzon 1990; Reutter et al. 2015a].

4.3. Navigational queries in practice

Next we show examples of how navigational queries can be expressed in practical query languages. As before we illustrate this using SPARQL, Cypher and Gremlin.

SPARQL. Since version 1.1 [Harris and Seaborne 2013], SPARQL permits the use of *property paths*, which are an extended form of regular expression that, beyond usual RPQs, also allow inverses and a limited form of negation [Kostylev et al. 2015b]. As a consequence, we can express any path query from Example 4.2 using SPARQL 1.1.

Example 4.7. Consider the RDF graph depicted in Figure 8. To find all pairs of actors who have finite collaboration distance (i.e. the query Q' from Example 4.1) we can use the following SPARQL query:

```
SELECT ?x ?y
WHERE { ?x (:acts_in/^:acts_in)* ?y }
```

Here the symbol '/' is used to denote concatenation and '^' to denote the inverse of an edge label. The Kleene closure is given by '*' as before. Note that if we wanted to extract the actors with a finite Bacon number from our graph database we can just replace the variable `?x` with the constant `:Kevin_Bacon`. □

In one aspect, SPARQL goes beyond RPQs and allows for a (very) limited form of negation called negated property sets [Kostylev et al. 2015b]. This is done by allowing subexpressions of the form $!\{e_1, \ldots, e_n\}$ inside property paths, which will match to all pairs of nodes connected by some edge whose label is not in the set $\{e_1, \ldots, e_n\}$. Apart from ordinary labels, negated property set can also include inverse-edge labels.

Example 4.8. Consider the RDF graph depicted in Figure 8 and the following SPARQL query with a negated property-set

```
SELECT ?y
WHERE { :Clint_Eastwood (!{:type,:directs})* ?y }
```

This query will match `:Unforgiven` (the IRI) and `"Unforgiven"` (the title string) for `?y`. Here, `:Anna_Levine` is not included since the negated property-set does not include any inverse. However, once any inverse is added, then inverse edges are included:

```
SELECT ?y
WHERE { :Clint_Eastwood (!{:type,:directs,^directs})* ?y }
```

This query will additionally return `:Anna_Levine` since now inverse edges are also traversed. In a similar manner to the first query, if the negated property set only includes inverses, then only inverse edges are traversed. □

Adding this limited form of negation to the RPQ-style features of property paths does not affect the complexity of SPARQL query evaluation [Kostylev et al. 2015b].

As aforementioned in the discussion on set vs. bag semantics in Section 4.1, in a draft of the SPARQL 1.1 standard, the original semantics of property paths was based on simple paths with a bag semantics. However, since it was shown that such a semantics quickly renders query evaluation impractical [Arenas et al. 2012; Losemann and Martens 2013], the semantics was changed. Now, in order to evaluate any query containing the transitive closure operator ($*$ or $+$), SPARQL uses a set semantics, looking for pairs of nodes connected by any path whose label belongs to the language of the regular expression specifying the query. Otherwise, if a property path can be rewritten as a bgp (with projection), SPARQL instead uses the bag semantics defined for bgps (see [Harris and Seaborne 2013, §9.3] for more details).

Similarly, SPARQL can also express navigational graph patterns (ngps).

Example 4.9. The ngp from Example 4.4 – find all people with a finite Erdős–Bacon number – can be expressed in SPARQL as:

```
SELECT ?x
WHERE { ?x (:acts_in/^:acts_in)* :Kevin_Bacon . ?x (:author/^:author)* :Paul_Erdos . }
```

This query is a conjunction of two RPQs, where the symbol . denotes conjunction. □

Likewise, SPARQL can express complex navigational graph patterns (cngps).

Example 4.10. Referring back to Example 4.6, we can express an RDF version of the query for the posts recommended to Julie but not to John as follows:

```
SELECT ?x
WHERE {
 {
  { :Julie :knows+/:likes ?x . ?x :hasTag/:hasFollower :Julie . }
   MINUS
  { :John :knows+/:likes ?x . ?x :hasTag/:hasFollower :John . }
}
```

This query involves the difference of two cgps, creating a cngp. □

Cypher. While not supporting full regular expressions, Cypher still allows transitive closure over a single edge label in a property graph. On the other hand, since it is designed to run over property graphs, Cypher also allows the star to be applied to an edge property/value pair; however, this is again limited to a single repeated label/value.

Example 4.11. To compute the friend-of-a-friend relation in Cypher over the graph from Figure 5, we can use the following expression:

```
MATCH (x1:Person) -[:knows*]-> (x2:Person)
RETURN x1,x2
```

This expression selects pairs of nodes that are linked by a path completely labelled by `knows`. To do this, it applies the star operator $*$ over the label `knows`. □

Currently Cypher does not allow to apply the recursive operator $*$ over more complex expressions; thus, for example, we are not able to query for actors with a finite Bacon number over the property graph from Figure 3 (without changing the data to, e.g., give explicit co-star relations). This might change, however, in the near future.

Recall that Cypher uses the no-repeated-edge semantics for cgps; by default, Cypher uses the same semantics for path queries, thus returning all pairs of nodes connected by a path which does not repeat any edges. In fact, Cypher uses a bag semantics, so each pair of nodes will be duplicated for every such path connecting them in the data.

Example 4.12. Consider the graph from Figure 5 and the following query looking for any path (of arbitrary length) between two nodes:

```
MATCH (x1) -[*]-> (x2)
RETURN x1,x2
```

Here the operator $*$ signifies that the path is of arbitrary length and there is no restriction on edge labels. The output of this query will contain the pair (n_1, n_4) twice, as there are two distinct paths (that do not repeat an edge) from the node n_1 representing `Julie`, to the node n_4 representing the post with the content `I love U2`. □

However, Cypher also allows for returning a single shortest path connecting two nodes, or all shortest paths connecting them, allowing the user to declaratively change the semantics for evaluating paths within the query.

Example 4.13. If we wanted to find friends of friends of Julie in the example above and return only the shortest witnessing path, we could use the following query:

```
MATCH ( julie:Person {firstname:"Julie"} ),
p = shortestPath( (julie) -[:knows*]-> (x:Person) )
RETURN p
```

This will return a single shortest witnessing path. If we wanted to return all shortest paths, we could replace "`shortestPath`" with "`allShortestPaths`". □

In Section 3 we have seen how to specify basic graph patterns using Cypher. A restricted form of navigational patterns – only allowing the star operator on edge labels – are then supported by allowing path expressions inside basic patterns.

Example 4.14. Coming back to the social network from Figure 5, if we want to find all friends-of-friends of Julie that liked a post with a tag that Julie follows, we can use the following Cypher query:

```
MATCH (x1:Person {firstName:"Julie"}) -[:knows*]-> (x2:Person)
MATCH (x2) -[:likes]-> () -> [:hasTag] -> (x3)
MATCH (x3) -[:hasFollower]-> (x1)
RETURN x2
```

The first `MATCH` clause provides a path expression, which when joined with the bgps expressed in the latter two `MATCH` clauses, forms a navigational graph pattern (ngp). In fact, the query is an abbreviated version of the ngp depicted in Figure 10. □

Apart from (a restricted form) of RPQs and (c)ngps, Cypher also offers several unique features that make it useful when working with property graphs. First, Cypher allows for specifying the length of the path. For instance, in Example 4.11 we can change the edge-label constraint `[:knows*]` to `[:knows*2..7]` to specify that the path must traverse at least two and at most seven edges. Although this property is syntactic and can be simulated using regular expressions, adding counting to regular expressions is known to improve the succinctness of the language [Losemann and Martens 2013].

Another interesting feature available in Cypher is the ability to return paths.

Example 4.15. If we wanted to return all friends of friends of Julie in the graph from Figure 5, together with a path witnessing the friendship, we can use:

```
MATCH p = (:Person {name:"Julie"}) -[:knows*]-> (x:Person)
RETURN x, p
```

The variable `p` will be bound by the witnessing path and will return (in Cypher syntax):

```
+-----------------------------------------------------------+
| x       | p                                               |
+-----------------------------------------------------------+
| Node[2] | [Node[1],:knows[1],Node[2]]                     |
| Node[1] | [Node[1],:knows[1],Node[2],:knows[2],Node[1]] |
+-----------------------------------------------------------+
```

We assume that `Node[1]` corresponds to n_1 (aka. John), `knows[1]` corresponds to e_1, and so forth. Each path is a sequence $n_1e_1n_2e_2n_3 \dots n_{k-1}e_{k-1}n_k$ as discussed previously. Though not shown, in practice Neo4j will also return all attributes and values on each node and edge. No further paths are returned since they repeat an edge. □

Gremlin. Gremlin supports navigation by the use of `repeat`, which enables arbitrary or fixed iteration of any graph traversal. As per SPARQL, Gremlin uses the arbitrary path semantics for navigational queries. However, unlike SPARQL, Gremlin returns bags and not sets of answers. Therefore, when returning nodes, Gremlin might repeat the same pair of nodes multiple (potentially infinite) times, depending on how many paths conforming to the query exist between them, and similarly for paths (which are defined in Gremlin as sequences of nodes).

Example 4.16. Recall how we used the following Gremlin expression in Example 3.16 to obtain all co-stars of Clint Eastwood:

```
G.V().hasLabel('Person').has('name','Clint Eastwood')
  .out('acts_in').hasLabel('Movies')
    .in('acts_in').hasLabel('Person')
```

For a fixed-length iteration, we can use repeat and specify the number of times the repetition should be performed. For example, the following traversal looks for actors that are linked to Clint Eastwood by a path of length 2:

```
G.V().hasLabel('Person').has('name','Clint Eastwood').repeat(
  out('acts_in').hasLabel('Movies')
    .in('acts_in').hasLabel('Person')
).times(2)
```

If we want arbitrary traversal we can simply omit the `times` command; however, this effectively means *iterate an unbounded number of times*, and consequently we may never get anything out of this traversal. For this reason we use the `emit()` modulator for repeat, which forces the repeat process to output the nodes after each iteration.

```
G.V().hasLabel('Person').has('name','Clint Eastwood').repeat(
  out('acts_in').hasLabel('Movies')
    .in('acts_in').hasLabel('Person')
 ).emit()
```

This query iterates an unbounded number of times, but at the end of each repetition, the current nodes of the traversal are output for the query. □

Finally, Gremlin also supports returning complete paths as results.

Example 4.17. To find all co-star paths connecting Clint Eastwood to other actors (and himself), we can use the following query:

```
G.V().hasLabel('Person').has('name','Clint Eastwood').repeat(
  out('acts_in').hasLabel('Movies')
    .in('acts_in').hasLabel('Person')
).emit().path()
```

This query will then begin enumerating all paths per the call to `path()`. □

There are several other features of repeat that can modify the traversal and output. For example, the `emit()` command can include conditions, such as `emit(hasLabel('Person'))` to output only those nodes labelled `'Person'`. Gremlin also includes an `until()` operator, to provide while-loop-style repetition, for example, to stop when a particular node is reached.

4.4. Complexity of evaluating navigational queries

We now discuss the complexity of evaluating navigational queries.

Path queries. We concentrate on the complexity of evaluating RPQs, which has received considerable attention in the theoretical literature. This is relevant since RPQs form the basis of many path query languages. We study the problem with respect to the possible restrictions we mentioned before, focusing on the problems of checking if a path exists, or finding pairs of nodes connected by some path under set semantics:

— *Arbitrary paths:* Determining whether v can be reached from u by a path labelled in the regular expression L can be solved in linear time $O(|G| \cdot |L|)$ (see, e.g., [Wood 2012; Barceló 2013]). This bound can be achieved by using folklore algorithms based on *automata* techniques. Such techniques can also be reformulated to compute the set of all pairs of nodes that are linked by a path labelled in L in time $O(|G|^2 \cdot |L|)$. In the special case of an unconstrained path query $Q = x \xrightarrow{*} y$, we can simply perform a directed reachability analysis over G. This can be done in time $O(|G|)$ for a single pair of nodes, and in $O(|G|^2)$ to compute all pairs of linked nodes.
— *Shortest paths:* Applying reachability techniques that return shortest paths (e.g., breadth-first search) in combination with the previous automata-based algorithms, we obtain shortest paths witnessing the constraints stated by RPQs. In particular, computing the set of all pairs of nodes that are linked by a path labelled in L, and for each such pair a shortest path in G witnessing it, can be done in time $O(|G|^2 \cdot |L|)$.
— *No-repeated-node/edge paths:* Under such semantics, the complexity jumps: evaluation becomes NP-complete even in data complexity [Mendelzon and Wood 1989]. Tractable instances of the RPQ evaluation problem under these semantics can be found by either restricting regular expressions or the class of graph databases [Mendelzon and Wood 1989; Bagan et al. 2013], but it remains to be seen to what extent such restrictions are relevant in practice. The special case of $Q = x \xrightarrow{*} y$ can still be computed efficiently since any shortest path needs to be simple, and thus finding an unconstrained simple path amounts to finding a shortest path.

In summary, finding nodes connected by arbitrary paths or finding a shortest path satisfying an RPQ can be done in polynomial time, whereas considering simple paths, the problem becomes intractable. An open question then is if there are any practical scenarios in which the (intractable) simple path witness is really justified in terms of computational cost over finding (tractable) witnesses based on shortest paths.

Please note that the discussion thus far assumes the use of set semantics when returning pairs of nodes or paths. When considering bag semantics in such scenarios, assuming a no-repeated-node/edge semantics, the complexity of the problem is at least that of the problem of counting paths under the chosen semantics [Arenas et al. 2012; Losemann and Martens 2013]; in the general case, this leads to a significant leap in complexity for reasons discussed previously.

Navigational graph patterns. Recall that an ngp is a bgp where the edges can also be labelled by an RPQ, or the special symbol $*$ denoting an arbitrary path. Assuming we adopt a set semantics for paths, evaluating an ngp Q over a graph database G can be implemented as follows:

(1) First, each RPQ $x \xrightarrow{L} y$ that labels an edge of Q is evaluated over the graph database G, and for each pair (u, v) of nodes that are connected by a path labelled with L we add to G a new edge between u and v labelled with L.
(2) Second, we evaluate P over the graph we augmented in the first step, but now treating P as a bgp (that is, L-labelled edges in P must only match to L-labelled edges in G, and not to a pair of nodes connected by a path whose label is in L).

Therefore, ngp evaluation can be separated into independent phases of path query evaluation (step 1) and graph pattern evaluation (step 2). This helps understand the complexity of evaluating ngps better.

(1) *First, how costly is step 1, i.e., building the augmented graph?* Of course this depends on the semantics for path query evaluation we use. If we use a simple path interpretation, this process will be intractable, while if we apply an arbitrary/shortest path interpretation, we can construct the graph in time $O(|G|^2 \cdot |Q|)$.
(2) *Second, how expensive is step 2, namely, evaluating a bgp over a graph?* We know from Section 3 that this problem is NP-complete in general, but tractable for certain efficient classes of queries and tractable in data complexity. Likewise if we consider c(n)gps, as in the case of SPARQL, the same complexity arguments apply.

Note that if we consider a bag semantics for paths, the first step will not succeed since a graph is a set of edges, and duplicate edges will not be preserved in the augmented graph; we would need an alternative strategy to capture such duplicate edges. In any case, the problem of constructing the augmented graph is already intractable in the case of set semantics, and will likewise be intractable in the case of bag semantics.

5. FINAL REMARKS

Graph databases are becoming more and more important in industry, with new graph database engines and query languages being released in recent years. With this emerging variety of systems and languages, understanding the features that each brings, and the fundamental issues that arise as a product of their design choices, is becoming of increasing importance. In this survey, we have provided an overview of the developments in this area, bridging theory and practice in order to develop a categorisation of features that constitute a common core for graph query languages.

Feature categorisation. We started our review of the core aspects of graph query languages by first presenting two graph database models: the *edge-labelled graph model*, and the more elaborate *property graph model*. Thereafter we identified the two main core features that are common in all modern graph query languages: *pattern matching* and *navigation*. We think that these two forms of querying are at the heart of graph query languages, and thus any reader that is familiar with these two classes of queries – and the different options that one could consider with respect to both – should be qualified to understand the core of any modern graph query language.[12]

To categorise pattern matching features, we identified the class of basic graph patterns (bgps), which should arguably form the core of any graph query language, and are indeed present in all of the practical systems we reviewed. These can be further extended with operators such as projection, union, or optional, among others, giving rise to complex graph patterns (cgps). In terms of navigational queries, following both the research literature and the practical solutions currently available, we identify paths

[12]Of course, there are also a number of additional operators which can be considered for graph querying, such as different forms of aggregation, or graph transformations; however, these either do not add anything fundamentally new to the core features we identified, or are implementation specific and not well explored in the literature. We provide a brief overview of such features in the online appendix to our paper.

Table I. Semantics adopted for pattern matching in SPARQL, Cypher and Gremlin.

Language	supported patterns	semantics
SPARQL	all complex graph patterns	homomorphism-based, bags*
Cypher	all complex graph patterns	no-repeated-edges†, bags*
Gremlin	complex graph patterns without explicit optional‡	homomorphism-based, bags*

$*$: All languages support a distinct operator to enable set semantics.
†: Homomorphisms can be simulated with multiple MATCH commands; see Example 3.11.
‡: Optional can be emulated imperatively.

Table II. Semantics adopted for navigational queries in SPARQL, Cypher and Gremlin.

Language	path expressions	semantics	choice of output
SPARQL	more than RPQs*	arbitrary paths, sets†	boolean / nodes
Cypher	fragment of RPQs	no-repeated-edge‡, bags§	boolean / nodes / paths / graphs
Gremlin	more than RPQs	arbitrary paths$^{||}$, bags§	nodes / paths

$*$: SPARQL adds negated property sets; see Example 4.8.
†: In the case of SPARQL, set semantics applies only when the query can *not* be rewritten as a cgp (e.g., when it uses a $*$ operator); see [Harris and Seaborne 2013] for details.
‡: Cypher also allows to enable shortest-path semantics.
§: A distinct operator is supported to enable set semantics
||: In Gremlin, other semantics can also be enabled or otherwise emulated.

as the core of all navigational queries over graphs, and adopt the well studied notion of regular paths queries (RPQs) as the basis for navigating graphs. These can then be incorporated into bgps giving rise to navigational graph patterns (ngps), which themselves can be further extended with operators such as union, optional, etc., to create the notion of complex navigational graph patterns (cngps).

The choice of the appropriate semantics for each of these forms of queries has proven to be a non-trivial task, and there have been several proposals coming both from practice and from theory. For matching basic graph patterns we classified the main proposals for the semantics into two categories:

(1) *homomorphism-based:* matching the pattern onto a graph with no restrictions.
(2) *isomorphism-based:* one of the following restrictions is imposed on a match:
— *no-repeated-anything:* no part of a graph is mapped to two different variables,
— *no-repeated-node:* no node in the graph is mapped to two different variables,
— *no-repeated-edge:* no edges in the graph is mapped to two different variables.

On the other hand, for path queries one can consider: (a) arbitrary paths; (b) shortest paths only; (c) paths not repeating a node (aka. simple paths); and (d) paths not repeating an edge. For the case of path queries there is also the question of how should their output look like. The options here range from: (i) checking the existence of a path (boolean output); (ii) returning start/end nodes of a path; (iii) returning complete paths; and (iv) returning entire graphs. In the case of both graph patterns and path queries, one can chose if answers are returned as bags (where duplicate answers are returned per their multiplicity), or sets (only a single copy of each answer is returned).

To exemplify our categorisation, we have reviewed some of the key design choices made for SPARQL, Cypher and Gremlin: three of the currently most popular query languages used in graph database engines. Table I contains a summary of these choices for pattern matching, and Table II likewise for navigational queries. Of course, all three languages extend upon these core features presented; however, this core offers a good starting point to further formalise, study and understand these languages.

Throughout, we have also discussed the effects of such design choices on the computational complexity considering various types of semantics and various evaluation problems. With respect to SPARQL, Cypher and Gremlin, we can summarise the following known results in terms of computational complexity of query evaluation, where *PM* refers to Pattern Matching and *NQ* to Navigational Queries.

— In terms of complexity, by far the most studied language of the three is SPARQL.
  PM. It is known that the evaluation of bgps with projection is NP-complete and that the evaluation of cgps is PSPACE-complete [Pérez et al. 2009].
  NQ. Evaluating cngps remains within the same complexity class as cgps – PSPACE-complete – assuming the set-based semantics of property paths used in the final official version of the SPARQL 1.1 standard [Kostylev et al. 2015b].
— With respect to Cypher, less is understood. One complication, in particular, is the use of the no-repeated-edge semantics, which has not been well-studied.
  PM. While evaluating bgps with projection in Cypher directly relates to the subgraph-isomorphism problem (which is NP-complete), there are no results stating how the no-repeated-edge semantics might affect the evaluation of cgps, so it is not clear if the problem is as hard (or perhaps even harder) than in the case of SPARQL: all that we can directly conclude is that evaluating cgps under this default semantics in Cypher is NP-hard. However, as per Example 3.11, a homomorphism-based semantics can be emulated by using multiple `MATCH` patterns; assuming such a "trick" is used, then SPARQL-like cgps can be modelled and the complexity is PSPACE-hard.
  NQ. Stating formal results is complicated by the fact that Cypher has a no-repeated-edge semantics (rather than the more well-studied no-repeated-node semantics for simple paths), a bag semantics, and that it does not support full RPQ-style expressions. Little is known of the complexity of such features, however, some lower bounds can be easily inferred. For instance, evaluating path queries is already NP-hard due to the fact that Cypher allows path unwinding (see our online appendix for more details).
— With respect to Gremlin, the language is Turing-complete. However, if we only consider the core fragments discussed herein, we can make the following conclusions:
  PM. As per Table I, we can see that the semantics of Gremlin and SPARQL are almost equivalent; even excluding the imperative features needed to emulate optional patterns in Gremlin, evaluating cgps should still be PSPACE-complete.
  NQ. The study of navigational queries in Gremlin is complicated by the combination of potentially infinite arbitrary paths, the default bag semantics and the presence of features that go beyond RPQs. However, we can note that considering only RPQ-style expressions returning nodes (and not paths) with the default arbitrary-path semantics, the expressivity is equivalent to SPARQL and the complexity of evaluating cngps should thus be the same as for cgps: PSPACE-complete.

This discussion shows that there are various open questions in terms of the complexity associated with the design choices made, in particular, by the Cypher query language.

Uses of this survey. First, the categorisation of models, query features, semantics and results covered by this survey offer a useful guide to anyone who wishes to understand a graph query language – be it an existing such language or one yet to be proposed – not in terms of superficial issues like query syntax or minor variations in the graph model, but rather in terms of fundamental querying abilities, choice of semantics, and expressivity. Once the core of a language can be understood in this more abstract way, different languages can then be compared and contrasted in a similar,

foundational manner. We have provided such a comparison for the languages SPARQL, Cypher and Gremlin. Indeed, even though these languages run over different models and have completely different syntax, etc., by looking at Table I and Table II, one realises that the core of these languages is fundamentally rather similar. In the same fashion, using this survey as a guide, we could now compare a new proposal for a graph query language by abstracting the pattern matching and navigational capabilities of the language, and asking relevant questions such as: "*what is the semantics of pattern matching in this language?*"; or "*what type of navigational features does it include?*".

This growing diversity of graph database technologies moreover suggests that the time may come for further standardisation of graph query languages. While SPARQL has been formally standardised for RDF databases and has been well studied in the literature, many implementers have opted for custom graph database solutions and engines with custom languages, such as Neo4j with Cypher. Likewise, ad hoc standards like Gremlin have emerged in recent years and have been implemented by multiple vendors. However, unlike SPARQL, the semantics and complexity of languages like Cypher and Gremlin have not been studied. Looking to the future, one can thus expect standardisation efforts to rigorously define and characterise the properties of a general graph query language that takes into consideration the demands of the industry, much like the story for SQL, where core features are abstracted as the relational algebra. Of course, a query language may not always abide by a clean abstraction, as per the case of SQL which goes beyond the relational algebra in various ways, or the languages in Table I and Table II that are annotated with exceptions and support a variety of other features not covered. But yet, the exercise of abstracting languages into core features is a necessary task if one wants to create standards in terms of understanding which features are simply syntactic (i.e., redundant in terms of expressivity), what choice of semantics and features could be considered, and what effects such choices have with respect to achieving desirable computational guarantees in terms of evaluating queries in that language. A notable such example of this were the studies by Arenas et al. [2012] and Losemann and Martens [2013] on the complexity of property paths initially proposed in a SPARQL 1.1 draft, which lead to the semantics being changed in the final version. Likewise, we believe that this survey can serve as a useful guide for current and future standardisation processes involving graph query languages.

We also expect that our survey could serve to bridge the theory/practice gap, helping to port theoretical results about abstract languages (such as graph patterns or path queries) into real graph database engines, and also the other way around, helping to state the problems facing current graph engines in a formal manner. Indeed, we have seen multiple times in this survey that a seemingly innocuous change can have a drastic effect on computational complexity upon further examination; for example, we saw how an optional operator in cgps leads to a jump in complexity for query evaluation, or how having a no-repeated-node semantics can render path queries intractable, or (in the aforementioned case of SPARQL 1.1) how the combination of bag semantics and path queries can quickly become problematic. On the other hand, for example, we have also seen that bpgs with projection can be extended with a variety of useful features without a complexity jump in terms of query evaluation, including support for ngps with path expressions. But while the existing theory provides important insights, this survey also reveals gaps in the literature. To name a few examples: the problem of finding tractable subclasses of graph patterns that can be evaluated efficiently over no-repeated-edge semantics is almost unexplored; systems capable of returning paths need a way of representing a set of paths whenever it is infinite; and we have already noted the importance of a more rigorous theoretical formalisation of Cypher and Gremlin in order to determine the exact complexity of evaluating their queries and to un-

derstand their expressive power. In this respect, we believe that our survey can bring research questions from the practical world of graph database engines into theory.

Future directions. Our categorisation also opens interesting possibilities for further work in terms of surveying and classifying other important aspects of graph databases that are infeasible to cover in this survey with the necessary depth.

The first issue has to do with the implementation and optimisation of modern query languages. In surveying fundamental features, we have only dealt with such issues indirectly. A number of implementations of modern graph query languages using the features in this survey have emerged: in terms of some of the most prominent SPARQL engines that have been released, we can name 4store [Harris et al. 2009], BlazeGraph (formerly BigData [Thompson et al. 2014]), GraphDB (formally (Big)OWLIM [Bishop et al. 2011]), Jena [Wilkinson et al. 2003], and Virtuoso [Erling 2012]; with respect to property graphs, Neo4j [The Neo4j Team 2016] is one of the most popular engines, but one can also cite Titan [DataStax 2015] and OrientDB [Tesoriero 2013]. Collectively these engines implement a diverse range of indexing strategies, query planning methods, optimisations and ad hoc heuristics – with more proposed in the literature – sometimes borrowing directly from relational databases (e.g., [Wilkinson et al. 2003]), others being custom-designed for graphs (e.g., [Bishop et al. 2011; Tesoriero 2013]), and others still that intersect the relational and graph worlds (e.g., [Erling 2012; Paradies et al. 2015]). The analysis and classifications of all these implementation strategies is an important task that can benefit tremendously from our framework and would make for an interesting complementary survey in the future.

The second important line of work has to do with identifying the core of graph analytics and other operations more related to machine learning, or computing statistics over graphs. Currently these operations are not commonly compiled into query languages, but instead graph engines normally provide several data-access primitives such that users can implement their own algorithms within a programming environment: a direction in which Gremlin goes, for example. Furthermore, there has recently been a lot of work on domain-specific languages that can take care of particular sets of operations within a certain domain or scenario (see e.g. [Hong et al. 2012]). However, in the area of graph analytics, with different tasks ranging from computing weighted shortest paths to computing the PageRank matrix of an entire graph, we see the same diversity problem as with graph query languages: how to abstract the core (possibly declarative) features of such operations?

More pertinently for this survey, it is not clear where graph query languages start and graph analytics languages end: what is the overlap of features required, how do they complement and/or extend each other, etc. The graph database community has been slow in adopting graph analytics as a problem of study, but as the importance of these operations grow, we expect this to change in the next few years. We believe that the first goal of the community should be identifying a common core of the most widely used operations, just as we have done with graph query languages. It would also be interesting to understand how classical database-querying tasks compare to engines supporting the so-called vertex-centric programming model, such as Apache Giraph [Han and Daudjee 2015], GraphX [Xin et al. 2013] or Pregel [Malewicz et al. 2010]. This again goes in the direction of Gremlin, which as we have discussed has many elements that are similar to a declarative query language, but also encapsulates a more imperative style, being supported, for example by, the aforementioned Apache Giraph analytics framework. We have also recently seen the first efforts in designing a more declarative language in this context [Jindal and Madden 2014], and in the following years we expect more research in this direction.

Conclusion. Recent years have seen the re-emergence of graph databases as an important alternative to their more widely-established relational cousin, bringing with them a variety of new challenges, new demands, and new questions. In this survey, we have provided an overview of the fundamental query features that underlie such databases and have provided a categorisation that generalises much of these recent developments and offers a bridge to known theoretical results while raising some new questions. Of course, there are many open challenges facing graph databases in terms of standardising query languages, implementing and optimising engines for query evaluation, studying the theoretical properties of related problems, as well as evolving graph databases to meet emerging demands for graph analytics. We hope that this survey may serve as a useful guide for those involved in such efforts.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

Charu C. Aggarwal and Haixun Wang. 2010. *Managing and Mining Graph Data*. Springer.

Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. 2009. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.* 7, 2 (2009), 57–73.

Andrej Andrejev and Tore Risch. 2012. Scientific SPARQL: Semantic Web Queries over Scientific Data. In *International Conference on Data Engineering (ICDE)*. 5–10.

Renzo Angles. 2012. A comparison of current graph database models. In *Graph Data Management (GDM) (ICDE Workshop)*.

Renzo Angles and Claudio Gutierrez. 2008. The Expressive Power of SPARQL. In *The Semantic Web (ISWC)*. 114–129.

Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *ACM Comp. Surv.* 40, 1 (2008).

Renzo Angles and Claudio Gutierrez. 2016. The Multiset Semantics of SPARQL Patterns. In *The Semantic Web (ISWC)*. Springer, 20–36.

Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: a unified language for event processing and stream reasoning. In *World Wide Web (WWW)*. 635–644.

Apache TinkerPop. 2017. TinkerPop3 Documentation v.3.2.5. http://tinkerpop.apache.org/docs/current/reference/. (June 2017).

Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *World Wide Web (WWW)*. 629–638.

Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2014. Expressive languages for querying the semantic web. In *Principles of Database Systems (PODS)*. ACM, 14–26.

Marcelo Arenas and Martín Ugarte. 2016. Designing a Query Language for RDF: Marrying Open and Closed Worlds. In *Principles of Database Systems (PODS)*. 225–236.

Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A trichotomy for regular simple path queries on graphs. In *Principles of Database Systems (PODS)*. 261–272.

Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2010. C-SPARQL: a Continuous Query Language for RDF Data Streams. *Int. J. Semantic Computing* 4, 1 (2010), 3–25.

Pablo Barceló. 2013. Querying graph databases. In *Principles of Database Systems (PODS)*. 175–188.

Pablo Barceló, Gaëlle Fontaine, and Anthony Widjaja Lin. 2015. Expressive Path Queries on Graphs with Data. *Logical Methods in Computer Science* 11, 4 (2015).

Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012a. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4 (2012), 31.

Pablo Barceló, Leonid Libkin, and Juan L. Reutter. 2014. Querying Regular Graph Patterns. *J. ACM* 61, 1 (2014), 8:1–8:54.

Pablo Barceló and Pablo Muñoz. 2014. Graph logics with rational relations: the role of word combinatorics. In *Logic in Computer Science (LICS)*. 12:1–12:10.

Pablo Barceló, Jorge Pérez, and Juan Reutter. 2013. Schema mappings and data exchange for graph databases. In *International Conference on Database Theory (ICDT)*. ACM, 189–200.

Pablo Barceló, Jorge Pérez, and Juan L. Reutter. 2012b. Relative Expressiveness of Nested Regular Expressions. In *Alberto Mendelzon Workshop (AMW)*. 180–195.

Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. 2000. Formal-language-constrained path problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.

Robert Battle and Dave Kolas. 2012. Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. *Semantic Web* 3, 4 (2012), 355–370.

Meghyn Bienvenu, Diego Calvanese, Magdalena Ortiz, and Mantas Simkus. 2014. Nested Regular Path Queries in Description Logics. In *Knowledge Representation & Reasoning (KR)*.

Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes, and Axel Polleres. 2012. Mapping between RDF and XML with XSPARQL. *J. Data Semantics* 1, 3 (2012), 147–185.

Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. *Semantic Web Journal* 2, 1 (2011), 33–42.

Andre Bolles, Marco Grawunder, and Jonas Jacobi. 2008. Streaming SPARQL - Extending SPARQL to Process Data Streams. In *European Semantic Web Conference (ESWC)*. 448–462.

Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. 2015. Reasonable Highly Expressive Query Languages. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 2826–2832.

François Bry, Tim Furche, Bruno Marnette, Clemens Ley, Benedikt Linse, and Olga Poppe. 2009. SPARQLog: SPARQL with Rules and Quantification. In *Semantic Web Information Management – A Model-Based Perspective*. 341–370.

Peter Buneman. 1997. Semistructured Data. In *Principles of Database Systems (PODS)*. 117–121.

Peter Buneman, Susan B. Davidson, Gerd G. Hillebrand, and Dan Suciu. 1996. A Query Language and Optimization Techniques for Unstructured Data. In *SIGMOD International Conference on Management of Data (SIGMOD)*. 505–516.

Horst Bunke. 2000. Graph matching: Theoretical foundations, algorithms, and applications. In *Vision Interface*. 82–88.

Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. Containment of Conjunctive Regular Path Queries with Inverse. In *Knowl. Representation & Reasoning (KR)*. 176–185.

Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2002. Rewriting of Regular Expressions and Regular Path Queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 443–465.

Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2003. Reasoning on regular path queries. *SIGMOD Record* 32, 4 (2003), 83–92.

Chandra Chekuri and Anand Rajaraman. 2000. Conjunctive query containment revisited. *Theor. Comput. Sci.* 239, 2 (2000), 211–229.

Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. 2008. Fast Graph Pattern Matching. In *International Conference on Data Engineering (ICDE)*. 913–922.

Mariano P. Consens and Alberto O. Mendelzon. 1990. GraphLog: a Visual Formalism for Real Life Recursion. In *Principles of Database Systems (PODS)*. 404–416.

Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *SIGMOD International Conference on Management of Data (SIGMOD)*. 323–330.

Víctor Dalmau, Phokion G. Kolaitis, and Moshe Y. Vardi. 2002. Constraint Satisfaction, Bounded Treewidth, and Finite-Variable Logics. In *Constraint Programming (CP)*. 310–326.

DataStax. 2015. Titan Documentation. http://s3.thinkaurelius.com/docs/titan/1.0.0/. (2015).

Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8.

Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *International Conference on Database Theory (ICDT)*. 8–21.

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. 2011. Adding regular expressions to graph reachability and pattern queries. In *International Conference on Data Engineering (ICDE)*. 39–50.

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010a. Graph Pattern Matching: From Intractable to Polynomial Time. *PVLDB* 3, 1 (2010), 264–275.

Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. 2010b. Graph Homomorphism Revisited for Graph Matching. *PVLDB* 3, 1 (2010), 1161–1172.

Diego Figueira and Leonid Libkin. 2015. Path Logics for Querying Graphs: Combining Expressiveness and Efficiency. In *Logic in Computer Science (LICS)*. 329–340.

Valeria Fionda, Giuseppe Pirrò, and Mariano P Consens. 2015. Extended property paths: writing more SPARQL queries in a succinct way. In *AAAI Conference on Artificial Intelligence*.

George HL Fletcher, Marc Gyssens, Dirk Leinders, Dimitri Surinx, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. 2015. Relative expressive power of navigational querying on graphs. *Information Sciences* 298 (2015), 390–406.

Daniela Florescu, Alon Y. Levy, and Dan Suciu. 1998. Query Containment for Conjunctive Queries with Regular Expressions. In *Principles of Database Systems (PODS)*. 139–148.

Riccardo Frosini, Andrea Calì, Alexandra Poulovassilis, and Peter T. Wood. 2017. Flexible query processing for SPARQL. *Semantic Web* 8, 4 (2017), 533–563.

César A. Galindo-Legaria and Arnon Rosenthal. 1997. Outerjoin Simplification and Reordering for Query Optimization. *ACM Trans. Database Syst.* 22, 1 (1997), 43–73.

Brian Gallagher. 2006. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium*. 43–53.

Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Paula Gearon, Alexandre Passant, and Axel Polleres. 2013. SPARQL 1.1 Update. W3C Recommendation. (21 March 2013). https://www.w3.org/TR/sparql11-update/.

Birte Glimm and Chimezie Ogbuji. 2013. SPARQL 1.1 Entailment Regimes. W3C Recommendation. (21 March 2013). https://www.w3.org/TR/sparql11-entailment/.

Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *Principles of Database Systems (PODS)*. 57–74.

Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB* 8, 9 (2015), 950–961.

Steve Harris, Nicholas Lamb, and Nigel Shadbolt. 2009. 4store: The Design and Implementation of a Clustered RDF Store. In *Scalable Semantic Web Knowledge Base Systems (SWSS)*. 94–109.

Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. (2013). http://www.w3.org/TR/sparql11-query/

Olaf Hartig. 2009. Querying Trust in RDF Data with tSPARQL. In *European Semantic Web Conference (ESWC)*. 5–20.

Olaf Hartig and Bryan Thompson. 2014. Foundations of an Alternative Approach to Reification in RDF. *CoRR* abs/1406.3399 (2014).

Pavol Hell and Jaroslav Nesetril. 2004. *Graphs and Homomorphisms*. Oxford University Press.

Jelle Hellings, Bart Kuijpers, Jan Van den Bussche, and Xiaowang Zhang. 2013. Walk logic as a framework for path query languages on graph databases. In *International Conference on Database Theory (ICDT)*. 117–128.

Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *36th Annual Symposium on Foundations of Computer Science*. 453–462.

Daniel Hernández, Aidan Hogan, and Markus Krötzsch. 2015. Reifying RDF: What Works Well With Wikidata?. In *Scalable Semantic Web Knowledge Base Systems (SWSS)*. 32–47.

David A. Holland, Uri Jacob Braun, Diana Maclean, Kiran-Kumar Muniswamy-Reddy, and Margo I. Seltzer. 2008. Choosing a data model and query language for provenance. In *2nd International Provenance and Annotation Workshop (IPAW)*.

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, 349–362.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2003. *Introduction to Automata Theory, Languages, and Computation – International Edition (2. ed)*. Addison-Wesley.

Alekh Jindal and Samuel Madden. 2014. GRAPHiQL: A graph intuitive query language for relational databases. In *Big Data*. IEEE, 441–450.

Graham Klyne, Jeremy J. Carroll, and Brian McBride. 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. (25 Feb. 2014). https://www.w3.org/TR/rdf11-concepts/

Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015b. SPARQL with Property Paths. In *The Semantic Web (ISWC)*. 3–18.

Egor V. Kostylev, Juan L. Reutter, and Martín Ugarte. 2015a. CONSTRUCT Queries in SPARQL. In *International Conference on Database Theory (ICDT)*. 212–229.

Manolis Koubarakis and Kostis Kyzirakos. 2010. Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL. In *European Semantic Web Conference (ESWC)*. 425–439.

Thomas Kurz, Kai Schlegel, and Harald Kosch. 2015. Enabling access to Linked Media with SPARQL-MM. In *World Wide Web (WWW), Companion*. 721–726.

Georg Lausen, Michael Meier, and Michael Schmidt. 2008. SPARQLing constraints for RDF. In *Extending Database Technology (EDBT)*. 499–509.

LDBC. 2015. LDBC Task Force: Property Graphs Data Model. http://www.ldbcouncil.org. (2015).

Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14.

Leonid Libkin, Juan Reutter, and Domagoj Vrgoč. 2013. Trial for RDF: adapting graph query languages for RDF data. In *Principles of Database Systems (PODS)*. ACM, 201–212.

Leonid Libkin and Domagoj Vrgoč. 2012. Regular path queries on graphs with data. In *International Conference on Database Theory (ICDT)*. 74–85.

Lorenzo Livi and Antonello Rizzi. 2013. The graph matching problem. *Pattern Anal. Appl.* 16, 3 (2013), 253–283.

Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.* 38, 4 (2013), 24.

Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. 2014. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.* 39, 1 (2014), 4:1–4:46.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 135–146.

Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. 2003. An efficient pathway search using an indexing scheme for RDF. *Genome Informatics Series* (2003), 374–375.

Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *Very Large Data Bases (VLDB)*. 185–193.

Robin Milner. 1989. *Communication and concurrency*. Prentice Hall.

Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

Hiroyuki Ogata, Wataru Fujibuchi, Susumu Goto, and Minoru Kanehisa. 2000. A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters. *Nucleic acids research* 28, 20 (2000), 4021–4028.

Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. 2015. GRAPHITE: an extensible graph traversal framework for relational database management systems. In *Scientific and Statistical Database Management (SSDBM)*. 29:1–29:12.

Task Force: Property Paths. 2009. Use cases in Property Paths Task Force. http://www.w3.org/2009/sparql/wiki/TaskForce:PropertyPaths#Use_Cases. (2009).

Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009).

Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2010. nSPARQL: A navigational language for RDF. *J. Web Sem.* 8, 4 (2010), 255–270.

Matthew Perry and John Herring. 2012. GeoSPARQL – A Geographic Query Language for RDF Data. Open Geospatial Consortium Implementation Standard. (2012). http://www.opengeospatial.org/standards/geosparql

Matthew Perry, Prateek Jain, and Amit P. Sheth. 2011. SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries. In *Geospatial Semantics and the Semantic Web – Foundations, Algorithms, and Applications*. 61–86.

Axel Polleres, Juan L. Reutter, and Egor V. Kostylev. 2016. Nested Constructs vs. Sub-Selects in SPARQL. In *Alberto Mendelzon Workshop (AMW)*.

Eric Prud'hommeaux and Carlos Buil-Aranda. 2013. SPARQL 1.1 Federated Query. W3C Recommendation. (21 March 2013). http://www.w3.org/TR/sparql11-federated-query/.

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

Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. 2015. TriAL-QL: Distributed Processing of Navigational Queries. In *Web and Databases (WebDB)*. ACM, 48–54.

Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2015a. Regular Queries on Graph Databases. In *International Conference on Database Theory (ICDT)*. 177–194.

Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. 2015b. Recursion in SPARQL. In *The Semantic Web (ISWC)*. 19–35.

Patrick Reynolds. 2015. Oracle of Bacon. http://www.oracleofbacon.org/. (2015).

Kaspar Riesen, Xiaoyi Jiang, and Horst Bunke. 2010. Exact and Inexact Graph Matching: Methodology and Applications. In *Managing and Mining Graph Data*. 217–247.

Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph Databases* (first ed.). O'Reilly Media.

Sebastian Rudolph and Markus Krötzsch. 2013. Flag & check: Data access with monadically defined queries. In *Principles of Database Systems (PODS)*. ACM, 151–162.

Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In *International Conference on Database Theory (ICDT)*. 4–33.

Jiwon Seo, Stephen Guo, and Monica S Lam. 2015. Socialite: An efficient graph query language based on datalog. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1824–1837.

Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB* 4, 11 (2011), 992–1003.

Claudio Tesoriero. 2013. *Getting Started with OrientDB*. Packt Publishing Ltd.

The Neo4j Team. 2016. The Neo4j Manual v3.0. http://neo4j.com/docs/stable/. (2016).

Bryan B. Thompson, Mike Personick, and Martyn Cutcher. 2014. The Bigdata® RDF Graph Database. In *Linked Data Management*. 193–237.

Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42.

Leslie G. Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.

Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a Property Graph Query Language. In *Graph Data Management Experiences and Systems (GRADES)*.

Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Symposium on Theory of Computing (STOC)*. 137–146.

Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, Dave Reynolds, and Luping Ding. 2003. Supporting Scalable, Persistent Semantic Web Applications. *IEEE Data Eng. Bull.* 26, 4 (2003), 33–39.

Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.

Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *Graph Data Management Experiences and Systems (GRADES)*. ACM, 2.

Xpath 1999. XML Path Language (XPath). http://www.w3.org/TR/xpath. (1999).

Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. 2016. A Short Survey of Recent Advances in Graph Matching. In *Multimedia Retrieval (ICMR)*. 167–174.

Jeffrey Xu Yu and Jiefeng Cheng. 2010. Graph Reachability Queries: A Survey. In *Managing and Mining Graph Data*. Advances in Database Systems, Vol. 40. Springer, 181–215.

Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia. 2012. A general framework for representing, reasoning and querying with annotated Semantic Web data. *J. Web Sem.* 11 (2012), 72–95.

Lei Zou, Lei Chen, and M. Tamer Özsu. 2009. DistanceJoin: Pattern Match Query In a Large Graph Database. *PVLDB* 2, 1 (2009), 886–897.

Online Appendix to: Foundations of Modern Query Languages for Graph Databases[13]

RENZO ANGLES, Universidad de Talca & Center for Semantic Web Research
MARCELO ARENAS, Pontificia Universidad Católica de Chile & Center for Semantic Web Research
PABLO BARCELÓ, DCC, Universidad de Chile & Center for Semantic Web Research
AIDAN HOGAN, DCC, Universidad de Chile & Center for Semantic Web Research
JUAN REUTTER, Pontificia Universidad Católica de Chile & Center for Semantic Web Research
DOMAGOJ VRGOČ, Pontificia Universidad Católica de Chile & Center for Semantic Web Research

Throughout the survey we focused on fundamental graph querying features as established by the research literature and used in practical graph database engines. In this appendix, we examine some emerging aspects of graph query languages, such as alternative semantics for basic graph patterns; navigational queries that go beyond paths; solution modifiers, such as aggregation, path unwinding and graph-to-graph queries; as well as some further extensions of SPARQL, Cypher and Gremlin. We start with the alternative semantics for BGPs.

A. ALTERNATIVE SEMANTICS FOR BASIC GRAPH PATTERNS

There are two main criticisms of the homomorphism and isomorphism based semantics. First, as discussed in Section 3.4, the computational complexity of key problems associated with these semantics can be quite high since they directly capture notions of graph homomorphism and subgraph isomorphism [Ullmann 1976] (which are both known to have NP-complete decision problems). Second, the matches defined by the above semantics are rigid, in the sense that they require the entire query to be matched onto the graph continuously. That is, even when all parts of the query can be matched to (possibly different parts of) the graph, they may return zero answers. To remedy the situation, one can deploy the more flexible notion of *graph-simulations* [Milner 1989] when defining a match, which gives rise to an additional semantics.

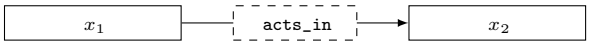
Simulation-based semantics: A generalisation of the notion of a homomorphism-based match has been proposed in the form of *graph-simulations* [Milner 1989], which, intuitively speaking, allow matching one node of a pattern to several nodes in the graph, as long as the structure of the pattern is preserved. Given an edge-labelled graph $G = (V, E)$ and a bgp $Q = (V', E')$, a simulation between Q and G is a relation $S \subseteq V' \times V$ such that: (i) for every node n' in V' there is a node n in V such that the pair (n', n) is in S, and (ii) for every pair $(n', n) \in S$ and every edge (n', r', m') in E', there exists an edge (n, r, m) in E such that $(m', m) \in S$ and $r = r'$ if $r' \in$ ***Const*** (and can be any value when $r' \in$ ***Var***). Then an answer to Q over G under the simulation-based semantics is any simulation S between Q and G. As shown in the literature, simulation-based semantics is computationally lighter for certain problems [Henzinger et al. 1995; Fan et al. 2011] and is more versatile when handling large graphs that might contain incomplete information [Fan et al. 2010a; Fan 2012; Fan et al. 2010b].

Simulation-based semantics can be naturally extended to property graphs. In this case, if a query node (or edge) uses constants in its attributes, we also require that it matches a graph node with equal values in the corresponding attributes. That is, when (v', v) belongs to our simulation, we also need the following conditions: (iii) if $\lambda(v') = r$ with $r \in$ ***Const***, then $\lambda(v) = r$; and (iv) $\sigma(v', e') = a'$, then $\sigma(v, e) = a$, for some e and

DOI: 0000001.0000001

a, with $e = e'$ when $e' \in \mathit{Const}$ and $a = a'$ when $a' \in \mathit{Const}$. A similar condition is also required for edge properties when specified in the query.

Example A.1. Consider again the graph G from Figure 1, and let Q be the following BGP:



One simulation between the query Q above and the graph G is given by the relation $S = \{(x_1, \texttt{Clint Eastwood}), (x_2, \texttt{Unforgiven})\}$. Another simulation is given by the relation $S' = \{(x_1, \texttt{Clint Eastwood}), (x_2, \texttt{Unforgiven}), (x_1, \texttt{Anna Levine})\}$, which in a sense contains matches for both Clint Eastwood and for Anna Levine. This exemplifies the fact that simulation-based semantics can capture multiple homomorphic matches in a single relation, which is one of the reasons why it can be evaluated more efficiently. □

The idea of matching the same query node to multiple graph nodes may be counter intuitive, as it captures "too much" information in a single relation. For this reason simulation-based semantics is often viewed as a base semantics for defining a set of "candidate matches" that can be further restricted and refined for particular use-cases, as has been explored recently by Ma et al. [2014].

The evaluation problem for the fragment consisting of bgps and projection can be solved in polynomial time for the case of the simulation-based semantics considered in Section 3.1 [Fan et al. 2010a].

B. MORE EXPRESSIVE NAVIGATIONAL QUERIES

For the navigational languages we have seen in Section 4, paths are the only form of recursion allowed, but to express certain types of queries, we may require more expressive forms of recursion. Imagine for instance that as in Example 4.2 we wish to check for all pairs of actors in our movie database that are connected by co-star relations, but only considering actors that have directed a movie (such as Clint Eastwood). We cannot express this query by a regular expression over paths since, aside from finding paths between co-stars, we need to check that each intermediate node in the path has an outgoing edge labelled `directs`. In this section, we present several languages that can express these types of queries, and explain how this can be achieved.

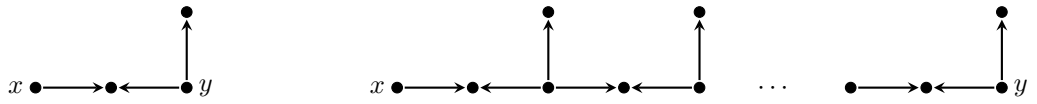
B.1. Nested regular expressions

The language of *nested regular expressions* (NREs) extends RPQs with a *branching* or *nesting* operator that allows to recursively check other nested RPQs over the nodes of a path. As such, the evaluation of an NRE consists of paths where nodes have a potentially branching path that satisfies the given nested RPQ. Conceptually speaking, NREs thus allow for capturing paths matched by a tree-shaped pattern, offering an increase in expressive power that has been applied in practice, for example, to form the basis of proposed navigational query languages for RDF [Pérez et al. 2010; Barceló et al. 2012b].

Example B.1. In the language of NREs, we can restrict our co-star paths to only consider directors using the following expression:

$$x \xrightarrow{\left(\texttt{acts\_in}\cdot\texttt{acts\_in}^{-}\,[\texttt{directs}]\right)^{+}} y\,.$$

This query asks for a path whose label belongs to the regular expression (with inverse) $(\texttt{acts\_in} \cdot \texttt{acts\_in}^{-})^{+}$, but imposes an additional condition: every intermediate node captured by the sub-expression $\texttt{acts\_in}\cdot\texttt{acts\_in}^{-}$ must have an outgoing edge labelled `directs`. More generally, the latter bracketed expression is an RPQ that is used as an



(a) Base for repetitions (b) Transitive closure of the base pattern

Fig. 11. Base of an NRE and the transitive closure over this base. We assume all horizontal edges in the above images to be labelled with `acts_in`, and all vertical edges with `directs`.

existential branching test on the preceding sub-expression, checking to ensure that each matched node is connected to some other node by the given bracketed expression. Note that the above pattern does not check that the start node is a director.

This recursive pattern is defined by the structure depicted in Figure 11: one can also think of this structure as taking the base pattern from Figure 11(a) and applying it recursively as illustrated in Figure 11(b). □

Just as we did with regular path queries, one can consider conjunctions of such patterns to arrive at the language of *conjunctive nested regular expressions* (CRNEs), which has thus far only been studied in theory [Barceló et al. 2013; Bienvenu et al. 2014]. Another direction to extend NREs is to add more expressive features such as negation and unary formulas. By doing so one arrives at a language that is equivalent to applying XPath [Xpath 1999] over graph databases. In fact, as shown by Libkin et al. [2016], NREs themselves correspond to a positive fragment of XPath.

B.2. Regular data path queries

While considering NREs, it is perhaps natural to consider how similar such patterns could be applied to property graphs, and in particular, to test the values of various node and edge attributes appearing along the path that one is traversing. To illustrate the issue, consider the following example.

Example B.2. Coming back to our social network, recall that in Example 4.1 we showed how to compute the friend-of-a-friend relation using the expression $\texttt{knows}^+$. Assume we now want to again compute the friend-of-a-friend relation, but we wish to consider only the people who live in the same country: each time we traverse an edge labelled `knows`, we need to check that the value of the country attribute is equal for both nodes connected by the edge. This can be expressed as follows:

$$e := ([\texttt{knows}]_{\texttt{start.country=end.country}})^+$$

where the filter `start.country=end.country` checks the aforementioned condition on all pairs of nodes connected by the `knows` edges that form the path. Note that unlike NREs, here we can express *comparisons* between the values of attributes. Also note that the brackets [] have different meaning in NREs as opposed to RDPQs. Namely, in the former they apply to nodes, and in the latter to the start/end point of a path the expression between the brackets defines. □

Expressions such as e above can be formalised by extending the grammar of the ordinary regular expressions with the operator $[exp]_c$, where exp is an expression, and c is a filter of the form $\texttt{start.atr=end.atr}'$, with `atr` and $\texttt{atr}'$ being attribute names. The full grammar is then given by the following $e := \ a \ ; \ e \cdot e \ ; \ e \,|\, e \ ; \ e^* \ ; \ [e]_c$, with c a conjunction of expressions of the form $\texttt{start.atr} = \texttt{end.atr}'$ or $\texttt{start.atr} \neq \texttt{end.atr}'$. Allowing any such expression e inside a path query $x \xrightarrow{e} y$ gives rise to *regular data path queries* (RDPQs), with the name signifying that paths consider not only navigational aspects, but also reason about the data stored in the graph.

Although queries that allow reasoning about how the attribute values change along paths seem to be relevant in practical applications, they seem to be poorly supported in existing systems. On the other hand, they did receive some attention in the theoretical literature. For instance, the base language for regular data path queries was introduced in [Libkin and Vrgoč 2012], and some further extensions allowing first order reasoning over paths [Hellings et al. 2013], or unlimited use of variables [Libkin et al. 2016; Barceló et al. 2015] have also been considered. However, since there is still no clear consensus on the correct language for this task, this seems to be a promising area of future work, both with respect to the theoretical issues, and with respect to the correct techniques for implementing such queries in graph database systems.

B.3. Datalog variants

Thus far all recursive navigational expressions we have considered are based on paths (e.g., RPQs) or trees (e.g., NREs). So what happens when we consider more general queries which look for repetitions of arbitrary bgps? It turns out that such queries can typically be expressed in *Datalog*-like languages [Abiteboul et al. 1995], which correspond to powerful recursive languages based on rules.

Example B.3. To exemplify how this works, let us focus on edge-labelled graphs (a similar translation can be devised for property graphs). Now instead of considering actors that are connected simply on merit of having co-starred in a movie, let us add the constraint that they must additionally direct a movie together (possibly a different movie). Let us call a pair of actor–directors connected (directly) in such a fashion "peers". Taking an edge-labelled graph G (in the style of Figure 1), we can create a query for peers as follows: $Q = (V, E)$, where $V = \{x, y, m, n\}$ are variables and where E contains $(x, \texttt{acts\_in}, m)$, $(x, \texttt{directs}, n)$, $(y, \texttt{acts\_in}, m)$ and $(y, \texttt{directs}, n)$.

To express this in Datalog we adopt the convention that the relation $\mathsf{E}(x, y, z)$ encodes an edge (x, y, z) in an edge-labelled graph $G = (V, E)$. We can then represent the original bgp Q as the following Datalog rule:

$$\mathsf{Q}(x, y) \leftarrow \mathsf{E}(x, \texttt{acts\_in}, m), \mathsf{E}(x, \texttt{directs}, n), \mathsf{E}(y, \texttt{acts\_in}, m), \mathsf{E}(y, \texttt{directs}, n)\,.$$

Applying this rule generates a binary relation Q that contains precisely the matches of bgp Q over G; in other words, we can quite easily represent a bgp as a Datalog rule and evaluate it as such.

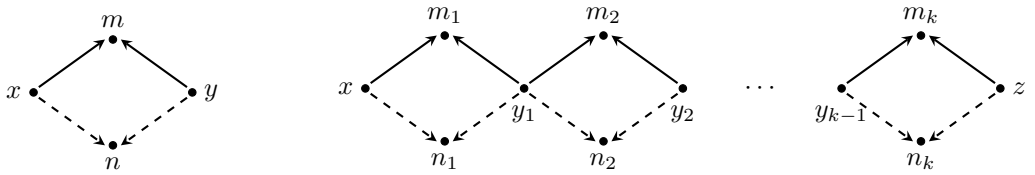
Let us assume we now wish to find all nodes connected recursively through a peer relation. We can add the following rule:

$$\mathsf{Q}(x, z) \leftarrow \mathsf{Q}(x, y), \mathsf{Q}(y, z)\,.$$

Applying these two Datalog rules in a recursive fashion generates an output Q that contains the transitive closure over peers.

More importantly – as illustrated in Figure 12 – the base pattern of Figure 12(a) is not a path nor a tree, and hence the resulting recursive pattern of Figure 12(b) achieved by these two Datalog rules would not be expressible in any language we discussed earlier: with Datalog, the recursive pattern can be an arbitrary bgp. □

In a manner analogous to returning paths for RPQs, one could consider trying to return a similar result for Datalog, but where instead of having sequences of nodes connected by edges in the case of RPQs, we would, intuitively speaking, have something more like sequences of sub-graphs in the case of Datalog. However, since the output of applying Datalog rules is a set of fixed-arity relations, it is not possible to return such a sequence; in fact, how to represent the structures that Datalog navigates is an unexplored area. Instead, Datalog rules can be applied to find pairs of nodes that



(a) Basic Datalog pattern (b) Recursive Datalog pattern (k is arbitrary)

Fig. 12. Illustration of the types of patterns a Datalog programs can query. Here all solid edges are labelled `acts_in` and all dashed edges are labelled `directs`.

are connected in such a manner, or to generate a relational representation of a graph that contains all such edges navigated, and so forth.

There have been attempts to define Datalog-like languages that are specifically tailored to the requirements of graph database applications, in the spirit of the recursive rules used in Example B.3. The first of these was GraphLog [Consens and Mendelzon 1990], which was designed for querying graphs formed by hypertext documents. More recently, Reutter et al. [2015a] studied the restriction of Datalog where recursion is only allowed over patterns that output at most two variables; in fact, a number of languages have been proposed in different settings with similar expressive power [Fletcher et al. 2015; Libkin et al. 2013; Rudolph and Krötzsch 2013; Arenas et al. 2014; Bourhis et al. 2015]. There have also been attempts to implement query engines that support these languages, specifically over RDF datasets using extensions of SPARQL [Reutter et al. 2015b; Przyjaciel-Zablocki et al. 2015]. Recently we have also witnessed proposals that combine user-defined functions into Datalog to obtain a graph query language more tailored for graph analytic tasks (see e.g. [Seo et al. 2015]).

In summary, the use of Datalog-like languages for querying graphs is an active area of research being explored from a number of angles. However, as we discuss in the following section, recursively applying bgps in a declarative manner is not widely supported within the practical query languages we consider in this survey.

B.4. Complexity considerations

When analysing more expressive variants of path queries, the evaluation complexity is deeply connected with the structure of the language. For languages such as NREs or XPath we can find fast evaluation algorithms that are nothing more than extensions of the algorithms shown for RPQs.

Concerning Datalog-based languages, it is well-known that answering unrestricted Datalog queries is EXPTIME-complete [Abiteboul et al. 1995]. Hence in practical settings, restrictions with lower complexity are sometimes considered. One such restricted language is Linear Datalog [Consens and Mendelzon 1990], for which query evaluation is PSPACE-complete. Other languages such as e.g. Regular Queries [Reutter et al. 2015a], may bound the arity of predicates, which returns query evaluation to the same complexity class as ngps: NP-complete.

Finally, with respect to regular data path queries of Section B.2, it can be shown that the base algorithm for RPQs can be modified in order to give a polynomial time evaluation [Libkin et al. 2016]. On the other hand, extending such queries with more expressive features seems difficult, as evaluation quickly becomes intractable [Libkin et al. 2016; Barceló et al. 2015]. Furthermore, implementing these queries using the unwinding operator, as in Cypher, does also not seem to be the best solution, as the operator makes the evaluation NP-hard (see Appendix C for details).

C. ADDITIONAL QUERY FEATURES

Throughout the survey our main focus is on querying features designed to retrieve nodes, edges, or paths from a graph. However, most practical query languages also include ways to manipulate these results, in particular, aggregating them or transforming them into different structures. While the types of operators offered for the manipulation of results vary significantly amongst different graph query languages, there are some common features in these languages that we explore in this section. In particular we look at aggregation functions, path manipulation and graph-to-graph querying functionalities, and discuss some challenges when implementing these operations over graphs.

C.1. Aggregation and solution modifiers

In the development of relational databases, the possibility of grouping values and computing statistics over these groups has been recognised as an important feature. In the case of SQL, the `GROUP BY` operator allows for grouping values according to some criteria, the `COUNT` operator allows for counting the number of elements in each such group, and the `MIN`, `MAX`, `SUM` and `AVG` operators were included to compute the minimum, maximum, sum and average of the elements in each group, respectively (provided that the group contains values compatible with the operators). These functionalities play such an important role in data analysis that they have been adopted by graph query languages. In what follows, we provide some examples of these features for the practical graph query languages considered in this survey, which will give the reader a clearer idea of how they are used.

Example C.1. As a first example, assume we have an edge-labelled graph G storing information about movies and actors, such as the one shown in Figure 8 on page 15. In order to count the total number of movies in G, we can use the following SPARQL query:

```
SELECT COUNT(?movie) AS ?total
WHERE { ?movie :type :Movie . }
```

As explained in Section 3, the triple `?movie :type :Movie` in this query is used to bind the variable `?movie` to the movies occurring in G. The operator `COUNT(?movie)` is then used to count the number of values for the variable `?movie`, which is stored in the variable `?total` as indicated by the command `COUNT(?movie) AS ?total`. If the variable `?movie` contains repeated values (which could happen in more complicated queries), then by default, duplicates will be counted; to ensure that only distinct values are counted, the command `COUNT(?movie)` can be replaced by `COUNT(DISTINCT ?movie)`. □

Example C.2. As a second example, assume that for each movie we wish to count the number of people acting in it. This query can be formulated as follows in SPARQL:

```
SELECT ?movie COUNT(DISTINCT ?actor) AS ?number_actors
WHERE {
        ?movie :type :Movie .
        ?actor :acts_in ?movie . ?actor :type :Person .
}
GROUP BY ?movie
```

The three triples inside the `WHERE` clause are used to indicate that for each pair b, c of values assigned to `?movie` and `?actor`, respectively, b must be a movie and c must be a person who acted in b. Then the operator `GROUP BY ?movie` is used to indicate that a group must be created for each value b in the variable `?movie`, which must contain all values c for the variable `?actor` such that b, c is a valid assignment for `?movie`

and `?actor` according to the triples in the `WHERE` clause. Finally, for each value b in `?movie`, the operator `COUNT(DISTINCT ?actor)` counts the number of distinct values in the group associated to b, which is stored in the variable `?number_actors` as indicated by the command `COUNT(DISTINCT ?actor) AS ?number_actors`. □

Example C.3. Assume now that each movie includes a property that defines its runtime. With such information we would like to obtain the longest films in the database. This query can be expressed as follows in Cypher:

```
MATCH (m:Movie) WITH MAX(m.runtime) AS maxTime
MATCH (m:Movie) WHERE m.runtime = maxTime
RETURN m
```

The first `MATCH` clause looks for nodes labelled `Movie` and stores them in variable `m`. The list of movies saved in `m` is explored by the `WITH` operator to compute the maximum runtime. From this first match clause, only the result of the aggregation (`maxTime`) can be projected. The second `MATCH` clause is thus needed to return the movies whose `runtime` is equal to the `maxTime` returned by the first `MATCH`. The filtered list of movies – movies with the longest runtime – is returned as the final result of the query. In this case, we say that the pattern initiated by the first `MATCH` clause is a *sub-query*.[14] □

All of the above examples can similarly be expressed in Gremlin.

Finally we briefly note that many practical query languages allow for applying *solution modifiers* over results, such as to express a limit for a number of results, or an ordering to apply over results, or an offset that specifies an number of initial results to skip. These solution modifiers can also be embedded within *sub-queries* that project the modified solutions to an outer query.

Example C.4. We can achieve a similar result to Example C.3 by instead using a solution modifier that orders by runtime and selects the first result:

```
MATCH (m:Movie) RETURN m ORDER BY m.runtime LIMIT 1
```

In this case, we require only one `MATCH` clause. However, this is not precisely equivalent to Example C.3: if we have multiple movies tied for the longest runtime, here we will only return one such movie, while previously we would return all such tied movies. To make the query equivalent, we would instead need a sub-query as follows:

```
MATCH (m:Movie) WITH m.runtime as maxTime ORDER BY maxTime LIMIT 1
MATCH (m:Movie) WHERE m.runtime = maxTime
RETURN n
```

As before, we use a sub-query to match any movie with the longest runtime and then find other movies with the same runtime. Note that unlike Example C.3 and the `MAX` aggregate, we could replace `LIMIT 1` with `SKIP 2 LIMIT 1` to find movies with the third longest runtime (where we could also replace `WITH` as `WITH DISTINCT` to filter ties). □

Such solution modifiers are also found in SPARQL and Gremlin.

As one can see, when coupled with basic graph patterns, aggregate operations and solution modifiers have a similar behaviour as in relational databases. On the other hand, when we consider navigational queries, such operations impose some unique challenges not present when dealing with relational data. For example, counting paths or taking the length of individual paths both impose computational challenges when

[14] It may seem counter-intuitive to have the sub-query "outside" in Cypher, as in SPARQL the first `MATCH` corresponds to a sub-query and would rather be written inside; as such, this is an idiosyncrasy of Cypher.

applied in this new context, as were raised in Section 4.1 when discussing the related problem of returning nodes from a path under bags semantics: if the graph database G is cyclic, the number of paths can be infinite and paths may have infinite length; on the other hand, while restrictions such as no-repeated-nodes make the set of paths finite, counting paths is still associated with a high computational complexity [Valiant 1979; Arenas et al. 2012; Losemann and Martens 2013]. Still, languages such as Cypher provide aggregation features that allow for counting such paths or taking their length.

Example C.5. Assume a graph database encoding a road network, where the connectivity between five cities (c_1, c_2, c_3, c_4 and c_5) is given by five (bidirectional) routes ($c_1 \leftrightarrow c_2$, $c_1 \leftrightarrow c_3$, $c_2 \leftrightarrow c_4$, $c_4 \leftrightarrow c_5$ and $c_3 \leftrightarrow c_5$). The longest route between cities c_1 and c_5 can be expressed in Cypher by the following query:

```
MATCH p = (a:City {name:"c1"})-[*]->(b:City {name:"c5"})
WITH MAX(length(p)) AS maxLength
MATCH p = (a:City {name:"c1"})-[*]->(b:City {name:"c5"})
WHERE length(p) = maxLength
RETURN p
```

In this example, the `MATCH` clause is used twice to store all the paths between cities c_1 and c_5 in variable `p` (since the sub-query can only return the result of the aggregate; see Example C.3). The `WITH` clause combines the operators `MAX` and `length` to obtain `maxLength`, i.e., the length of the longest path. The `WHERE` clause selects paths whose length is equal to `maxLength`. The final result is the list of the longest paths such that each path is encoded as a collection of nodes and edges, which in this case would be:

```
[{name: c1}, {}, {name: c2}, {}, {name: c4}, {}, {name: c5}]
```

This is the longest path from city c_1 to city c_5 without a repeated edge. Note that without the restriction on repeating edges, we could have infinite length paths (for example, subsequently going back and forth between c_3 and c_5 ad infinitum). □

In Cypher, we can also, for example, count all paths.

Example C.6. Consider a query that counts the number of paths from a source node to a target node in a graph. This query is expressed in Cypher as follows:

```
MATCH p = (:A)-[*]->(:B)
RETURN COUNT(p)
```

The `MATCH` clause in this query stores the paths from a node with label `A` to a node with label `B` in the variable `p`, and the `COUNT` clause counts the number of paths stored in `p`; again the no-repeated-edges restriction avoids infinity in the case of cycles. □

While in Cypher, the restriction of not repeating edges is offered by default, in Gremlin, a call to `simplePath()` is required to ensure that nodes are not repeated.

Example C.7. The following Gremlin query computes all paths between nodes with labels `A` and `B` such that no path visits the same node twice:

```
G.V().hasLabel('A').repeat(out().simplePath()).until(hasLabel('B')).emit().path()
```

The `simplePath()` function filters paths that repeat nodes. Interestingly, Gremlin returns paths ordered by ascending length, and thus by keeping only the first answer we can use this query to obtain the shortest path. □

As opposed to the case of relational aggregates, many questions about aggregation functions on paths remain open. In particular, understanding the expressive power of

these functions and pinpointing the exact complexity of evaluating them are important open issues that deserve further investigation.

C.2. Path unwinding

Path unwinding refers to the idea of projecting parts of a path. As previously discussed, SPARQL queries cannot return paths: they can either check for the existence of paths satisfying some conditions, or return the set of start- and/or end-nodes of such paths; thus, SPARQL does not provide any path-ungrouping operator. On the other hand, Cypher provides functions to get path elements independently.

Example C.8. Recall the road network described in Example C.5. When travelling between cities c_1 and a city c_5, we may wish to find two different disjoint routes (visiting disjoint intermediate cities) allowing us to see new scenery on each part of our journey. A query finding two such paths can be expressed in Cypher as follows:

```
MATCH p1 = (a:City {name:"c1"}) -[*]- (b:City {name:"c5"})
MATCH p2 = (a:City {name:"c1"}) -[*]- (b:City {name:"c5"})
WHERE none(x IN nodes(p2) WHERE (x IN nodes(p1) AND x<>a AND x<>b))
RETURN p1, p2
```

The variables `a` and `b` store the nodes representing the cities with names `c1` and `c5`, respectively. The variables `p1` and `p2` then store paths between these two cities; these paths are undirected and of arbitrary length as indicated by the expression `-[*]-`. The expression `nodes(path)` returns the nodes in the path as a collection, while `relationships(path)` returns the edges in the path as a collection. The path disjointness condition is defined by using the `none` operator (which evaluates to `true` if the condition is false for all elements of a collection). In more detail, the `WHERE` clause specifies that for two paths `p1` and `p2` to be returned, there can be no node `x` such that: (i) `x` is a node in the path `p2` as indicated by the condition `x IN nodes(p2)`; (ii) `x` is a node in the path `p1` as indicated by the condition `x IN nodes(p1)`; and (iii) `x` is different from `a` and `b` as indicated by the condition `x<>a AND x<>b`. In other words, `p1` and `p2` are returned only if they do not share any nodes aside from `a` and `b`. □

Although useful, queries such as the one above are inherently difficult to evaluate. In fact, given a graph G and nodes b and c in G, the problem of verifying whether there exist two paths in G between b and c with no nodes in common except for b and c is known to be NP-complete (this problem is referred as the two-disjoint-paths problem in the literature [Garey and Johnson 1990]). Hence we see that adding path unwinding to a query language can lead to issues with computational complexity when combined with other features of the language: various well-known hard problems can be trivially expressed using such combinations of features.

Gremlin has similar features for path unwinding, where nodes and edges can be extracted from paths and processed with subsequent operators.

C.3. Graph-to-Graph queries

Both the input and output of an SQL query are relational tables, so this language is compositional in the sense that the output to a query can be used as the input of another query. Along similar lines, graph query languages provide functionalities that allow to return a graph as the result of a query.

In the case of SPARQL, the `SELECT` operator can be replaced by the `CONSTRUCT` operator in order to produce an RDF graph as the output of a query. More specifically, a SPARQL query of the form `CONSTRUCT { t1 t2 ... tn } WHERE { ... }` produces an RDF graph as output, where each t_i is an RDF triple that can contain variables and constants, and where the `WHERE` clause is defined as usual. To produce the answer to

such a query, first the WHERE clause is evaluated to produce all possible matches. Next, each match is applied to replace the variables occurring in t_1, t_2, ..., t_n by constants. A match may not have a value for a variable occurring in a specific triple t_i because of the use of the operators OPTIONAL and UNION; in this case, an output RDF triple is not produced from t_i for that match. Finally, RDF graphs are defined as unordered sets, meaning that duplicates and ordering are not preserved in the output graph.

Example C.9. Take again the RDF graph in Figure 8 on page 15, which we denote by G. To create an RDF graph G' storing information about people that act together in some movie, we can use the following query:

```
CONSTRUCT { ?actor1 :act_together ?actor2 . }
WHERE {
   ?movie :type :Movie .
   ?actor1 :acts_in ?movie . ?actor2 :acts_in ?movie .
   FILTER (?actor1 != ?actor2)
}
```

For each assignment b, c_1, c_2 generated by evaluating the WHERE clause for the variables ?movie, ?actor1, ?actor2, respectively, we have that c_1 and c_2 act together in the movie b, and also that c_1 and c_2 are distinct actors as indicated by the command FILTER (?actor1 != ?actor2). This assignment replaces ?actor1 by c_1 and ?actor2 by c_2 in the CONSTRUCT clause to produce the triple c_1 :act_together c_2. In the case of Figure 8, we would thus create a new RDF graph with two edges labelled :act_together connecting :Clint_Eastwood to :Anna_Levine, and vice versa. □

In the case of Cypher, one can include a CREATE clause inside a query expression to create graph elements (nodes and edges) from the pattern matching step.

Example C.10. Consider the property graph with movie data from Figure 3 on page 6. Similarly to Example C.9, if we want to construct a graph containing only the pairs of actors that co-starred in a movie, we can use the following Cypher query:

```
MATCH (a:Person)-[:acts_in]->(:Movie)<-[:acts_in]-(c:Person)
WHERE a <> c
CREATE (a)-[r:act_together]->(c)
RETURN r
```

The CREATE clause will then "materialise" the graph containing all pairs of actors that co-starred in the same movie. The RETURN clause then specifies that all of the edges of this graph should be returned. We also add a WHERE clause to distinguish a from c: although Cypher adopts a no-repeated-edge semantics, there may be multiple edges from an actor to a movie, for example, if the actor plays multiple roles in that movie, in which case we would generate vacuous loops on such actors in the output. □

A similar mechanism for graph creation is provided by Gremlin.

Example C.11. Consider now a transportation network that connects two cities if there is a direct bus link between them. Suppose we want to travel, but are only willing to change the bus once. To see our options, we could add an edge labelled twoHoplink between any two cities reachable from each other by at most one change of bus. This can be done using the following Gremlin query:

```
G.V().as("a").out().out().as("b").addE("twoHoplink").from("a").to("b")
```

In the above expression: G.V().as("a") obtains the list of nodes in the graph and store this list in variable a; .out().out().as("b") obtains the nodes b reachable from each node in a, considering a single intermediate node on this path;

`addE("twoHoplink").from("a").to("b")` creates edges labelled `twoHoplink` between each pair of connected nodes stored in `a` and `b`. □

The graph-to-graph queries illustrated in the examples above are a rather new feature for graph query languages; currently there are few studies about their basic properties and the effects of combining them with other query features. Some work involving the expressive power and the composition of queries using `CONSTRUCT` in SPARQL has been carried out [Kostylev et al. 2015a; Polleres et al. 2016; Arenas and Ugarte 2016]. However, the use of these types of queries in Cypher or Gremlin is currently unexplored in the literature, and may be an interesting topic for future research.

D. FURTHER EXTENSIONS

A number of extended features have been proposed and/or included in the SPARQL, Cypher and Gremlin languages. Though the focus of this survey is on the core features of graph matching and navigational queries, we give a brief overview of some of the more prominent extensions, both as included in the respective specifications of the query languages themselves, and as proposed by third parties in the literature.

With respect to official extensions, SPARQL 1.1. Update [Gearon et al. 2013] defines a specification for making updates to the underlying dataset that the SPARQL engine queries, allowing to add, remove or modify graphs or triples with graphs in a declarative manner; likewise Cypher offers primitives to update nodes, edges and the labels and attributes associated with them [The Neo4j Team 2016], while Gremlin supports updates through use of the Blueprints API that forms part of the TinkerPop framework [Apache TinkerPop 2017]. In order to standardise a mechanism for processing queries over data spanning multiple sources, SPARQL 1.1 Federated Query [Prud'hommeaux and Buil-Aranda 2013] specifies how SPARQL queries can contain nested queries that are sent to and executed by remote SPARQL services, with the results returned to the outer query for further local processing. With respect to reasoning, SPARQL 1.1 Entailment [Glimm and Ogbuji 2013] provides details on how various types of ontological and rule-based entailment regimes can be applied to generate further answers from implicit knowledge during the graph matching process.

Aside from official extensions, a wide variety of extensions have been proposed by third parties in the research literature, particularly for the SPARQL language. Various such extensions are concerned with supporting additional meta-information for RDF data: two such proposals are SPARQL* [Hartig and Thompson 2014] and AnQL [Zimmermann et al. 2012], which both describe general frameworks for *reifying* or *annotating* RDF data (respectively), providing analogous query features in SPARQL. Other general extensions of interest include SPARQL^{AR} [Frosini et al. 2017], which allows for performing query approximation and relaxation to also return "near answers"; SPARQLog [Bry et al. 2009], which extends SPARQL with rules and more flexible forms of quantification, additionally defining fragments that maintain desirable complexity results; XSPARQL [Bischof et al. 2012], which allows for federating queries over SPARQL, XML (through XQuery) and relational databases (through SQL) in a unified manner; as well as work by Lausen et al. [2008] on using SPARQL (and a proposed extension thereof) to specify relational-like constraints over RDF graphs.

Other proposed extensions of SPARQL target specific domains or types of applications, including tSPARQL [Hartig 2009], which allows for specifying and processing trust annotations in terms of which results can be trusted and why; SciSPARQL [Andrejev and Risch 2012], which provides primitives to deal with numeric arrays of (scientific) information; SPARQL-MM [Kurz et al. 2015], which

proposes user-defined functions to help when querying meta-data about multimedia artefacts; GeoSPARQL [Perry and Herring 2012; Battle and Kolas 2012], stSPARQL [Koubarakis and Kyzirakos 2010] and SPARQL-ST [Perry et al. 2011], which propose extensions to support spatial and temporal queries; EP-SPARQL [Anicic et al. 2011], C-SPARQL [Barbieri et al. 2010] and Streaming SPARQL [Bolles et al. 2008], which deal with processing dynamic information and support, offering event processing, reasoning and querying over windows of streaming data, and so forth.

The above discussion suggests that research in the areas of graph querying and analytics is ongoing, with various extended features being continuously proposed. Graph query languages are thus sure to evolve to capture more and more features. Rather than trying to cover all such features in detail, in this survey, we focus on capturing a core set of features that are foundational for querying graphs in a declarative manner and that thus form the common backbone of modern graph query languages.