Query Languages for Graph Databases

Peter T. Wood
Department of Computer Science and Information Systems, Birkbeck, University of London

ptw@dcs.bbk.ac.uk

ABSTRACT

Query languages for graph databases started to be investigated some 25 years ago. With much current data, such as linked data on the Web and social network data, being graph-structured, there has been a recent resurgence in interest in graph query languages. We provide a brief survey of many of the graph query languages that have been proposed, focussing on the core functionality provided in these languages. We also consider issues such as expressive power and the computational complexity of query evaluation.

1. INTRODUCTION

Graphs are widely used for representing data, with the result that a number of query languages for graphs have been proposed over the past few decades. In the 1980s motivating applications came from areas such as hypertext systems [17, 63]. When semi-structured data [2, 12] and object databases [34] became prominent in the 1990s, these provided fruitful areas for the study of graph models and query languages. In the last decade, the semantic web [9, 57] and social networks [5, 24, 60, 61] have taken over as key areas amenable to graph-based approaches. Further application areas for graph querying include transportation networks [11], semantic associations as part of criminal investigations [62] (also called link analysis), biological networks [46, 47, 48], program analysis [50], workflow and data provenance [6, 39].

Each of the above application areas has its own requirements in terms of an appropriate graph data model. In its simplest form a graph G is a pair (V, E), where V is a finite set of nodes and E is a finite set of edges connecting pairs of nodes. Of course, edges can be directed or undirected, although we will consider only the directed case here (which is more general). In most applications it is also the case that edges are labelled in some way, sometimes with sets of attribute-value pairs. Similarly, in general nodes may be labelled with sets of attribute-value pairs. However, we will mostly limit our discussion to graphs in which each node is identified by a distinct label (identifier) and each directed edge is labelled with a symbol drawn from some finite alphabet Σ; hence $E \subseteq V \times \Sigma \times V$.

Some application areas require graph structures that are more elaborate than the simple model described above. For example, hypergraphs have been used to model hypertext [63], while the hypernode model [58] allows for nodes that can themselves comprise graphs. In contrast, the so-called blobs of the Hy$^+$ system [19] comprise sets of nodes and share some similarities with the blobs of higraphs [35]. In addition, a number of graph data models require that each graph conforms to a *schema*. However, for the purposes of this paper, we will assume only the simple model described above; for details on more elaborate graph data models, we refer the reader to the survey by Angles and Gutiérrez [7].

Figure 1 shows an example of a graph G conforming to our simple definition, inspired by an example from NAGA [43, 64]. Node labels in G denote names of authors, literary prizes and locations. Edge labels denote the $hasWon$ relationship between authors and prizes (abbreviated w), the $bornIn$ relationship between authors and places (abbreviated b), the $livesIn$ relationship between authors and places (abbreviated i), and the $locatedIn$ relationship between places. Note that there can be paths comprising $locatedIn$ edges: for example, Bacchus Marsh is a town $locatedIn$ the state of Victoria which is $locatedIn$ the country Australia.

A typical query Q on graph G might ask to find authors who have won both the Booker and Nobel prizes—a simple *conjunctive query* (CQ) returning a set of nodes as answer. Query Q might be expressed using the following syntax

$$ans(x) \leftarrow (x, hasWon, Nobel), (x, hasWon, Booker)$$

where x is interpreted as a node variable, while

Database Principles Column. Column editor: Pablo Barceló, Department of Computer Science, University of Chile. E-mail: pbarcelo@dcc.uchile.cl.

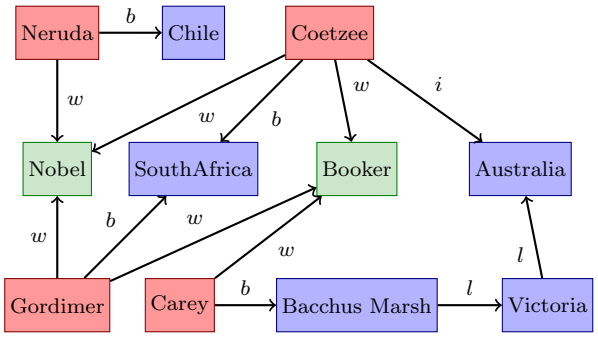

Figure 1: A graph of authors, prizes they have won, and places where they were born.

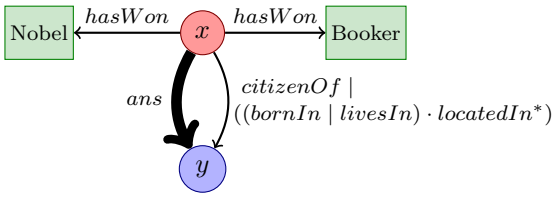

Figure 2: Query to find places related to authors who have won both the Nobel and Booker prizes.

hasWon, *Nobel* and *Booker* are constants. This syntax is reminiscent of Datalog, except that atoms in the body do not have predicate names since only a single graph is being queried; indeed, the atoms are similar to the *triple patterns* used in SPARQL [36], the W3C query language for RDF [44].

It is common in querying graphs that users may want to find pairs (x, y) of nodes such that there is a path from x to y whose sequence of edge labels matches some pattern. One way of specifying such a pattern is by using a *regular expression* defined over the alphabet of edge labels [52]. Such a query is called a *regular path query* (RPQ). So, using the example of Figure 1, an RPQ using the regular expression $citizenOf \mid ((bornIn \mid livesIn) \cdot locatedIn^*)$ asks for pairs of author x and place y such that x is a citizen of y or was born in or lives in y, or where x was born in or lives in some place which is connected to y by a sequence of any number of *locatedIn* relationships.

CQs and RPQs can be combined to form *conjunctive regular path queries* (CRPQs). For example, the following query Q adds the conjuncts from the example CQ to those of the RPQ as follows:

$$\begin{aligned} &ans(x, y) \\ &\quad \leftarrow \quad (x, hasWon, Nobel), (x, hasWon, Booker) \\ &\qquad\quad (x, (citizenOf \mid ((bornIn \mid livesIn) \cdot locatedIn^*)), y) \end{aligned}$$

CRPQs formed the basis of the languages **G** [21] and GraphLog [17], although those languages used a syntax of graph patterns. Figure 2 shows how the CRPQ Q above would be expressed in GraphLog, where there is an obvious mapping between the edges in the graph pattern and the atoms in the body of Q. The thick edge in Figure 2 is called the *distinguished* edge, representing edges which occur in the answer of the query; hence, it corresponds to the head of Q.

CRPQs were much studied with respect to querying semistructured data in languages such as Lorel [2], StruQL [26] and UnQL [13], as well as in terms of query containment [16, 28], query rewriting [15] and so on. CRPQs reappeared more recently in NAGA [43], for example, in a form similar to that in Figure 2. The ability to query paths using regular expressions, and hence provide the functionality of CRPQs, has only very recently been introduced in SPARQL 1.1 [36].

However, for a number of problems arising in graph querying CRPQs are insufficiently powerful [10]. These problems include comparing semantic associations in RDF graphs [8], comparing biological sequences [31], and so on. In such settings, we want to be able to express *relations* among paths. For example, the following query Q finds entities x and y such that the same sequence of edge labels π connects x and y as connects *Coetzee* and y:

$$(x, y) \quad \leftarrow \quad (Coetzee, \pi, y), (x, \pi, y), \Sigma^*(\pi) \quad (1)$$

Here, π is a *path variable*, and Σ^* denotes any sequence of edge labels. Q is an example of an *extended* conjunctive regular path query (ECRPQ), as proposed in [10].

We might also want to include paths themselves in the output of a query. This has been proposed, for example, as an extension to SPARQL [45], and is also provided by ECRPQs; to return the paths as part of the answer to query Q above, one simply includes the path variable π in the head. Of course, if there are cycles in the input graph, the answer to a query may be infinite. In such cases a compact representation of the set of answers to an ECRPQ can be returned in the form of an automaton [10]. ECRPQs are described in more detail in Section 3.3.

Requirements arising from querying and analysing social networks bring the need for further capabilities to be provided by graph query languages [5, 24, 60, 61]. In particular, aggregation functions play an essential role in network analysis, while the ability to transform networks by creating new nodes based on (aggregations of) sets of existing nodes is also crucial. We discuss these requirements further, and provide examples, in Sections 3.4 and 3.5.

With the proliferation of data on the Web (e.g., in the form of linked data), it is less likely that users will be familiar with the terms and structure used, which in any case will also be more heterogeneous. In such situations, queries that permit flexible [42, 51] or approximate matching [30, 40] of data may be helpful. We consider this further in Section 3.6.

After a brief survey of many graph query languages in the next section, we focus on the core functionality provided by such languages in Section 3. This section covers subgraph matching (Section 3.1), finding nodes connected by paths (Section 3.2), comparing and returning paths (Section 3.3), aggregation (Section 3.4), node creation (Section 3.5), and approximate matching and ranking (Section 3.6). Section 4 covers the expressive power of languages and the computational complexity of query evaluation. We conclude the paper in Section 5. We do not cover a number of other topics of interest such as query containment or query optimisation or evaluation in general.

2. A BRIEF SURVEY

In this section we give a brief overview of some of the graph query languages developed over the past 25 years or so. In particular, we highlight the different syntax used by various languages, as well as their proposed area of application. Section 3 discusses the functionality underlying these languages in more detail, while the expressive power and complexity of evaluating queries in some of the languages is presented in Section 4.

We have already mentioned the query languages **G** [21] and GraphLog [17]. The data model used by these languages is that of a labelled, directed graph. In G, a query is a set of pairs of graphs, each pair comprising a pattern graph and a summary graph. This pair of graphs essentially represents a CRPQ, with a set of such pairs being interpreted as disjunction. GraphLog replaced the summary graph with a distinguished edge, as shown in Figure 2. GraphLog also added edge inversion, negation and aggregation functions (Section 3.4), while defining a semantics different from that of **G**. The semantics of **G** was defined in terms of matching *simple* paths in the graph being queried (Section 3.2), whereas the meaning of a GraphLog query was given by the meaning of the stratified Datalog program to which it was translated (examples are given in Sections 3.2 and 3.4).

Other early graph query languages include GRAM [4] and GraphDB [32]. The data models of both require the presence of a graph schema. Both provide regular expressions defined over alternating sequences of node and edge types. GraphDB includes object-oriented features such as classes for nodes and edges, as well as paths. The intended area of application for GRAM was Hypertext, while that for GraphDB was spatial networks such as transportation systems. As a result, GraphDB provides built-in operators such as one for shortest path.

GOOD is another graph query language based on an object-oriented model [33, 34]. GOOD's querying mechanism is via graph transformations: node addition/deletion and edge addition/deletion. Also provided is an abstraction mechanism to group objects by means of their properties, as well as methods for defining sequences of operations. GOOD gave rise to a number of successor languages, such as G-Log [56] and the update language GUL [38].

A number of query languages were developed to query graphs represented in the Object Exchange Model (OEM) [55] or one of its variants/derivatives. OEM was developed to model semistructured data which had no predefined schema and could be heterogeneous. The Lore (Lightweight Object Repository) graph data model and its associated query language Lorel [2] distinguish two types of nodes: complex objects and atomic objects (values) which have no outgoing edges. A graph must have a number of named nodes (or entry points), and every node must be reachable from a named node.

In common with many graph query languages, Lorel uses a syntax based on OQL, allowing regular expressions over edge labels. A distinctive feature of Lorel is the availability of path variables.

Say we wish to formulate a Lorel query Q equivalent to the ECRPQ shown in (1). In Lore, each node has an oid rather than a label as in Figure 1, so authors' names, for example, would be represented by separate atomic objects connected to author nodes by an edge labelled *name*, say. We also assume that *Winners* is a named entry point, with edges labelled *author* to author nodes. Then Q can be written as

select X, Y
from *Winners.author* A, *Winners.author* X
$A.\#@P.Y$, $X.\#@Q.Y$
where $A.name$ = *'Coetzee'*
and *path-of*(P) = *path-of*(Q)

where # denotes a path of any length, so is equivalent to the regular expression Σ^* used earlier. $@P$ binds the path (of oids and labels) to variable P, and *path-of* returns a sequence of edge labels.

STRUDEL is another system whose data model is based on the OEM data model [26]. Its intented area of application is the implementation of data-intensive web sites, whose content and structure is specified using the query language STRUQL. The

language is compositional; the result of a query on a site graph is another site graph. Once again, regular path expressions are used, but because there is a need to create graphs corresponding to web sites, new constructs are needed. These include the `link` clause which creates a new graph from existing graphs, using Skolem terms for new nodes.

UnQL is a functional query language for semistructured data based on structural recursion [13]. The data model used somewhat different to that of OEM, being value-based rather than object-based. UNQL queries are translated to an internal algebra, UnCAL, which allows for optimisation. Once again, regular path patterns are provided, but the functional nature of UnQL means that graphs can be constructed, using data constructors, rather than only queried. UnQL's data model includes special symbols called *markers*, which are related to object identifiers in models such as OEM, except that not every node in a graph has to have a marker. Each graph also has certain nodes designated as inputs and certain nodes designated as outputs.

YAGO/NAGA combines database and information retrieval techniques to provide a semantic search engine for web derived knowledge [64]. The NAGA data model is a directed, weighted multigraph in which nodes represent entities, edges represent relationships, and weights represent confidence of extracted facts. A query is a connected, directed graph in which each edge is labelled with a regular expression over edge labels or a variable or the *connect* keyword (similar to Figure 2). A query using *connect* returns the paths connecting the corresponding nodes. Answers to queries are ranked in terms of informativeness, confidence and compactness (e.g., short paths rank higher than long paths).

SocialScope [5] aims to provide information discovery and presentation from social content sites such as Yahoo! Travel. To do this, it proposes a uniform algebraic framework operating on the social content graph, a graph in which both nodes and edges have attributes. The algebra provides operations of node selection, edge selection, graph union, intersection and difference, graph composition and semi-join, and aggregation functions for node aggregation and edge aggregation.

Other query languages for social networks include SoSQL [60], BiQL [24] and SNQL [61]. The two basic query structures in SoSQL are paths and groups (sets of nodes). Paths can have predicates and aggregate functions applied to them. Path predicates can include path operators such as *all* or *at most* n, specifying that all or at most n nodes or edges satisfy a given predicate. Groups can also have aggregation operators applied to them. BiQL [24] uses an SQL-like syntax to query and transform networks. New networks are formed using a `CREATE` clause. Aggregation functions can be used in the `WHERE` clause to restrict results, as well as in the `SELECT` clause to define new attribute values. SNQL [61] uses a data model comprising actors, relationships and attribute values (all represented as nodes), with edges associating attribute values with actors or relationships and associating actors with the relationships in which they participate. The query language is based on GraphLog, adding Skolem functions in order to create new nodes in the output. These new nodes are based on grouping or aggregating nodes or attribute values in the input (see Section 3.5).

3. QUERY LANGUAGE FUNCTIONALITY

In this section, we focus on the functionality provided by typical graph query languages. The following subsections will consider functionality in terms of the following broad categories: subgraph matching, finding nodes connected by paths, comparing and returning paths, aggregation, node creation, and approximate matching and ranking. Of course, many languages offer operations such as union (disjunction), composition and negation of queries, but we will not cover these separately.

We start with some general notation and definitions. Let $G = (V, E)$ be a graph as defined in Section 1. Given a query expression Q and a graph G, the evaluation of Q on G is denoted $Q(G)$.

Let us call the following question the *query evaluation problem* (QEP). Given a query expression Q and a graph G, is $Q(G)$ non-empty? As usual, one can consider the complexity of this problem by possibly fixing one of the two inputs. *Combined complexity* corresponds to when both Q and G are part of the input. *Query complexity* is when the input is Q, with G being fixed, while *data complexity* is when the input is G, with Q being fixed. We often consider data complexity to be the most relevant measure since graphs are assumed to be large and query expressions short.

3.1 Subgraph matching

In some sense, the simplest form of graph query supported by all languages is one which finds subgraphs within a graph. This corresponds to a *conjunctive query* (CQ). Let us fix a countable set of *node* variables (typically denoted by $x, y, z, \ldots$). A

We will from now on not usually distinguish between an expression in a query language and the query (function) it denotes, simply using the term "query" for both.

conjunctive query (CQ) Q over a finite alphabet Σ is an expression of the form:

$$ans(z_1, \ldots, z_n) \leftarrow \bigwedge_{1 \leq i \leq m} (x_i, a_i, y_i) \qquad (2)$$

such that $m > 0$, each x_i and y_i is a node variable or constant ($1 \leq i \leq m$), each $a_i \in \Sigma$ ($1 \leq i \leq m$), and each z_i is some x_j or y_j ($1 \leq i \leq n$, $1 \leq j \leq m$). The atom $ans(z_1, \ldots, z_n)$ is the *head* of the query, while the expression on the right of the arrow is its *body*. The query Q is *Boolean* if its head is of the form $ans()$, i.e. $n = 0$.

Let $\bar{x} = (x_1, \ldots, x_m)$, $\bar{y} = (y_1, \ldots, y_m)$ and $\bar{z} = (z_1, \ldots, z_n)$. The semantics of CQs Q of the form (2) are defined as follows. Let σ be a mapping from $\bar{x}, \bar{y}$ to the set of nodes of a graph $G = (V, E)$ which is the identity on constants. We define a relation $(G, \sigma) \models Q$ which holds iff $(\sigma(x_i), a_i, \sigma(y_i)) \in E$, for $1 \leq i \leq m$. Then $Q(G)$ is the set of tuples $\sigma(\bar{z})$ such that $(G, \sigma,) \models Q$. If Q is Boolean, we let $Q(G)$ be true iff $(G, \sigma) \models Q$ for some σ.

Using the example graph G from Figure 1, the following CQ finds authors born in South Africa who have won both the Nobel and Booker prizes:

$$\begin{array}{rcl} ans(x) & \leftarrow & (x, hasWon, Nobel), \\ & & (x, hasWon, Booker), \\ & & (x, bornIn, SouthAfrica) \end{array}$$

Each CQ of the form shown in (2) is formulated with respect to a single graph and returns a set of bindings for each node variable mentioned in the head of the query. However, in some application areas, the database to be queried comprises a *set* of graphs, and the answer to a query is the *subset* of graphs in which a match is found (e.g., in biological applications). Other languages allow single or multiple graphs to be queried and return the set of matching *subgraphs* [37, 43].

Although CQs are in some sense the simplest form of graph queries, they have been the subject of much study, particularly in terms of finding efficient ways of evaluating them on large graphs. This is because the combined complexity of the QEP for CQs is the same as the problem of subgraph ismorphism, which is well-known to be NP-complete. Because of this, Fan et al. have been investigating an alternative semantics for graph pattern matching based on graph simulation [25].

3.2 Finding nodes connected by paths

Let $G = (V, E)$ be a graph over alphabet Σ, with $v_0, v_m \in V$. A *path* ρ between nodes v_0 and v_m in G is a sequence $v_0 a_0 v_1 a_1 v_2 \cdots v_{m-1} a_{m-1} v_m$, where $m \geq 0$, $v_i \in V$ ($1 \leq i \leq m$), $a_i \in \Sigma$ ($1 \leq i < m$), and $(v_i, a_i, v_{i+1}) \in E$ ($1 \leq i < m$). The *label* of such a path ρ, denoted by $\lambda(\rho)$, is the string $a_0 \cdots a_{m-1} \in \Sigma^*$. The length of ρ is m. We also define the empty path as (v, ε, v) for each $v \in V$; the label of such a path is the empty string ε.

Regular path queries Determining reachability between nodes in a graph is a querying mechanism found in most graph query languages. The class of *regular path queries* [15, 21, 49, 52] provides queries which return all pairs of nodes in a graph connected by a path conforming to some regular expression. A *regular path query* (RPQ) Q is an expression of the form

$$ans(x, y) \leftarrow (x, r, y) \qquad (3)$$

where x and y are node variables, and r is a regular expression over Σ. Here we use $|$ for alternation (disjunction) and $\cdot$ for concatenation. We also allow the shorthands of r^+ for $(r \cdot r^*)$, $r?$ for $(r|\epsilon)$, and Σ for $(a_1| \cdots |a_n)$. We may also use a^- to match an edge labelled a in the *reverse* direction, i.e., from head to tail rather than from tail to head [16]. For example, the regular expression $citizenOf \mid ((bornIn \mid livesIn) \cdot locatedIn^*)$ is used in the example query in Figure 2.

Let G be a graph, r be a regular expression, and ρ be a path in G. Path ρ *satisfies* r if $\lambda(\rho) \in L(r)$, the language denoted by r. Given an RPQ Q of the form given in (3), the answer of Q on G, denoted $Q(G)$, is the set of all pairs of nodes (x, y) in G such there is a path from x to y which satisfies r.

The Regular Path Problem is, given a query Q of the form given in (3), a graph G and a pair of nodes x and y, is $(x, y) \in Q(G)$? It is well known that this problem can be solved efficiently [52]. One algorithm is as follows: (i) construct a nondeterministic finite automaton (NFA) M_r, with initial state s_0 and final state s_f, accepting $L(r)$; (ii) consider G as an NFA with initial state x and final state y; (iii) form the product automaton $M_r \times G$ of M_r and G; and (iv) determine whether there is a path from (s_0, x) to (s_f, y) in $M_r \times G$. Each step of this algorithm can be performed in Ptime, so the Regular Path Problem has Ptime combined complexity.

Alternatively, we can translate Q into a set of Datalog rules. So if Q uses the regular expression $citizenOf \mid ((bornIn \mid livesIn) \cdot locatedIn^*)$ from Figure 2, the translation might be as follows:

$$\begin{array}{lcl} ans(x, y) & \leftarrow & citizenOf(x, y) \\ ans(x, y) & \leftarrow & assoc(x, y) \\ ans(x, y) & \leftarrow & assoc(x, z), partOf(z, y) \\ assoc(x, y) & \leftarrow & bornIn(x, y) \\ assoc(x, y) & \leftarrow & livesIn(x, y) \\ partOf(x, x) & \leftarrow & locatedIn(x, x) \\ partOf(x, y) & \leftarrow & locatedIn(x, z), partOf(z, y) \end{array}$$

As an alternative to the semantics defined above, we might instead want to match only *simple* paths in G that satisfy the regular expression r. A path ρ is *simple* if no node is repeated on ρ. The REGULAR SIMPLE PATH PROBLEM can then be stated as, given a graph G, a pair of nodes x and y in G and a regular expression r, is there a *simple* path from x to y satisfying r? It turns out, however, that the REGULAR SIMPLE PATH PROBLEM is NP-complete, even for fixed regular expressions [52].

Conjunctive regular path queries Extending regular path queries by allowing conjunctions of atoms yields the class of *conjunctive* regular path queries [16, 28]. A *conjunctive regular path query* (CRPQ) Q over Σ is an expression of the form

$$ans(z_1, \ldots, z_n) \leftarrow \bigwedge_{1 \leq i \leq m} (x_i, r_i, y_i) \qquad (4)$$

in which all variables are as for a CQ, except that each r_i is now a regular expression over Σ. The query of Figure 2 corresponds to a CRPQ.

Let $\bar{x}$, $\bar{y}$ and $\bar{z}$ be defined as for CQs, and $G = (V, E)$ be a graph. The semantics of CRPQs Q of the form (4) are defined analogously to that for CQs, with σ being a mapping from $\bar{x}, \bar{y}$ to V. The relation $(G, \sigma) \models Q$ holds iff, for $1 \leq i \leq m$, there exists a path ρ_i in G from $\sigma(x_i)$ to $\sigma(y_i)$ such that $\lambda(\rho_i) \in L(r_i)$. Now $Q(G)$ is defined as for CQs.

3.3 Comparing and returning paths

As suggested in Section 1, there are a number in situations in which we may want to specify relations between paths as well as to have the actual path(s) connecting two nodes returned as query answers, whether to find connections in linked data on the web (DBPedia, Freebase), for analysis in social or other networks, or to determine data provenance.

Providing both of these capabilities gives rise to the class of *extended CRQPs* (ECRPQs), introduced in [10] from where most of the material in this subsection is derived. ECRPQs extend the class of CRPQs in two ways. Firstly, they allow *free path variables* in the heads of queries. Secondly, they permit checking *relations on sets of paths* in the bodies of queries, rather than simply conformance of individual paths to regular languages.

We first define the notion of *regular* relations over Σ, which are used in ECRPQs. Let $\perp$ be a symbol not in Σ. We denote the extended alphabet $(\Sigma \cup \{\perp\})$ by $\Sigma_\perp$. Let $\bar{s} = (s_1, \ldots, s_n)$ be an n-tuple of strings over alphabet Σ. We construct a string $[\bar{s}]$ over alphabet $(\Sigma_\perp)^n$, whose length is the maximum of the s_j's, and whose i-th symbol is a tuple $(c_1, \ldots, c_n)$, where each c_k is the i-th symbol of s_k, if the length of s_k is at least i, or $\perp$ otherwise. In other words, we pad shorter strings with the symbol $\perp$, and then view the n strings as one string over the alphabet of n-tuples of letters. An n-ary relation S on Σ^* is *regular* if the set $\{[\bar{s}] \mid \bar{s} \in S\}$ of strings over alphabet $(\Sigma_\perp)^n$ is regular (i.e., accepted by an automaton over $(\Sigma_\perp)^n$, or given by a regular expression over $(\Sigma_\perp)^n$). We shall often use the same letter for both a regular expression over $(\Sigma_\perp)^n$ and the relation over Σ^* it denotes, as doing so will not lead to ambiguity.

In additional to the set of node variables defined for CRPQs, we now also fix a countable set of *path* variables (denoted by $\pi, \omega, \chi, \ldots$). An *extended conjunctive regular path query* (ECRPQ) Q over Σ is an expression of the form:

$$ans(\bar{z}, \bar{\chi}) \leftarrow \bigwedge_{1 \leq i \leq m} (x_i, \pi_i, y_i), \bigwedge_{1 \leq j \leq t} R_j(\bar{\omega}_j), \quad (5)$$

such that

(i) $m > 0$, $t \geq 0$,

(ii) each R_j is a regular expression that defines a regular relation over Σ,

(iii) $\bar{x} = (x_1, \ldots, x_m)$ and $\bar{y} = (y_1, \ldots, y_m)$ are tuples of node variables,

(iv) $\bar{\pi} = (\pi_1, \ldots, \pi_m)$ is a tuple of distinct path variables,

(v) $\{\bar{\omega}_1, \ldots, \bar{\omega}_t\}$ are distinct tuples of path variables, such that each $\bar{\omega}_j$ is a tuple of variables from $\bar{\pi}$, of the same arity as R_j,

(vi) $\bar{z}$ is a tuple of node variables from $\bar{x}, \bar{y}$, and

(vii) $\bar{\chi}$ is a tuple of path variables from those in $\bar{\pi}$.

The semantics of ECRPQs is defined by a natural extension of the semantics of CRPQs. For an ECRPQ Q of the form (5), a graph $G = (V, E)$ and mappings σ from node variables to V and μ from path variables to paths, we write $(G, \sigma, \mu) \models Q$ if

- $\mu(\pi_i)$ is a path in G from $\sigma(x_i)$ to $\sigma(y_i)$, for $1 \leq i \leq m$, and
- for each $\bar{\omega}_j = (\pi_{j_1}, \ldots, \pi_{j_k})$, the tuple of strings consisting of labels of $\mu(\pi_{j_1}), \ldots, \mu(\pi_{j_k})$ belongs to the relation R_j.

The answer of Q on G is defined as

$$Q(G) \;=\; \{(\sigma(\bar{z}), \mu(\bar{\chi})) \mid (G, \sigma, \mu) \models Q\}.$$

We now present some examples, taken from [10]. In a query language for RDF/S introduced in [8], paths can be compared based on specific *semantic associations*. Edges correspond to RDF properties and paths to property sequences. A property a can

be declared to be a subproperty of property b, which we denote by $a \prec b$. Two property sequences u and v are called ρ-isomorphic if and only if $u = u_1, \ldots, u_n$ and $v = v_1, \ldots, v_n$, for some n, and $u_i \prec v_i$ or $v_i \prec u_i$ for every $i \leq n$ [8]. Nodes x and y are called ρ-isoAssociated iff x and y are the origins of two ρ-isomorphic property sequences.

Finding ρ-isoAssociated nodes cannot be expressed using a CRPQ, not least because doing so requires checking that two paths are of equal length. However, pairs of ρ-isomorphic sequences can be expressed using the regular relation R given by the expression $\left(\bigcup_{a,b\in\Sigma:\ (a\prec b \vee b \prec a)}(a,b)\right)^*$. An ECRPQ returning pairs of nodes x and y that are ρ-isoAssociated can then be written as follows:

$$ans(x,y) \leftarrow (x,\pi_1,z_1),(y,\pi_2,z_2),R(\pi_1,\pi_2)$$

Path variables in an ECRPQ can also be used to return the actual paths found by the query, a mechanism found in the query languages proposed in [2, 8, 39, 45]. For instance, SPARQLeR [45] introduces path variables and regular expressions into the SPARQL query language, allowing paths to be output. As an example, the SPARQLeR query that returns every path between the RDF resources r and s, provided the path includes the resource e, can be expressed by the ECRPQ

$$ans(\pi_1,\pi_2) \leftarrow (r,\pi_1,e),(e,\pi_2,s)$$

where π_1 and π_2 are the actual paths matched.

Regular expressions with backreferencing (REBRs) [3], as provided by *egrep* and Perl, for example, extend regular expressions by including expressions of the form $(r)\%X$, where r is a regular expression and X is a variable, which binds a string $w \in L(r)$ to X. Subsequent uses of X in the expression then match w. REBRs can denote non-regular languages [3]. Although ECRPQs can mimick some of this functionality over paths in a graph, it was recently shown that ECRPQs cannot express all REBRs [29]. On the other hand, ECRPQs can match patterns, such as $a^n b^n c^n$, where $a, b, c \in \Sigma$ and $n \in \mathbb{N}$, that cannot be denoted by REBRs, by using an equal-length (el) predicate as follows:

$$\begin{aligned} ans(x,y) \quad \leftarrow \quad & (x,\pi_1,z_1),(z_1,\pi_2,z_2),(z_2,\pi_3,y), \\ & a^*(\pi_1),b^*(\pi_2),c^*(\pi_3), \\ & \mathrm{el}(\pi_1,\pi_2),\mathrm{el}(\pi_2,\pi_3), \end{aligned}$$

where $\mathrm{el}(\pi,\pi')$ is shorthand for $(\bigcup_{a,b\in\Sigma}(a,b))^*(\pi,\pi')$.

3.4 Aggregation

Determining various properties of graphs requires computation that goes beyond matching and path finding. Such properties range from simple computations to determine the degrees of nodes to more complex ones for computing the eccentricity of a node, the distance between pairs of nodes, or the diameter of a graph. To formulate queries which return the values of such properties, we need aggregation operators such as *count*, *sum*, *min* and *max*.

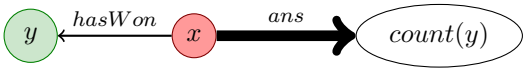

Figure 3: GraphLog query to count the number of prizes for each author.

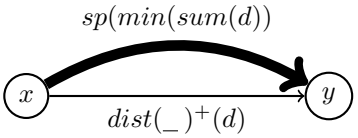

Figure 4: GraphLog shortest path query

Aggregation was available in early graph query languages such as $\mathbf{G}^+$ [22] and GraphLog [20]. It is also a feature of query languages for social networks, such as [24, 60, 61]. GraphLog and SNQL [61] have semantics that are based on Datalog with aggregation. For consistency with the rest of the paper, we will use the language of CRPQs extended with aggregation functions, denoted CRPQ^{agg}, without giving a formal definition.

In GraphLog [20], aggregate terms are allowed in the label of a distinguished edge or distinguished node. The simple query in Figure 3 counts, for each author x, the number of prizes y they have won. The equivalent CRPQ^{agg} is:

$$ans(x, count(y)) \quad \leftarrow \quad (x, hasWon, y)$$

In general, GraphLog and SNQL queries are translated into *recursive* Datalog programs, and combining recursion with aggregation can lead to nontermination. Hence, care is taken to ensure that the recursive rules that are produced perform transitive closure computations over closed semirings [18], such as formed by the operators *min* and *sum* in computing shortest paths, as illustrated below.

Finding the length of the shortest path between each pair of nodes requires that we summarise values (i.e., distances) along a path (by summing them) and then aggregate these summarised values (by finding the minimum). This is what the GraphLog query Q in Figure 4 does. The variable d in Q is a *collecting* variable, *sum* is a *summarising* function, and *min* is used to aggregate the summarised distances. Query evaluation is in PTIME if summarisation and aggregation operators form a closed semiring [18]. Query Q might be translated to the

following Datalog program [20]:

$$\begin{array}{lcl} len(x,x,x,0) & \leftarrow & dist(x,y,l) \\ len(x,x,x,0) & \leftarrow & dist(y,x,l) \\ len(x,z,y,d) & \leftarrow & sp(x,z,s), dist(z,y,l), d = s+l \\ sp(x,y,min(d)) & \leftarrow & len(x,z,y,d) \end{array}$$

Predicate $len(x, z, y, d)$ specifies that there is a path of length $d = s + l$ from x to y via z, where the length of the shortest path from x to z is s and the distance from z to y is l.

As mentioned in Section 2, SocialScope [5] provides node aggregation and edge aggregation. The result of aggregation can be represented as a new attribute of a node or edge in the graph. With the help of a user-defined function for defining similarity between users, [5] shows how aggregate functions can be used to express collaborative filtering on travel recommendations.

3.5 Node creation

In languages such as GraphLog the answer to a query is a graph, where the edges and their labels can be new, but the nodes are drawn from those of the graph being queried. However, in a number of applications, there is a need for the output of a query to contain nodes that were not part of the input. This requirement appears in web site management (and hence in StruQL [26]) and in social network analysis and transformation (and hence in BiQL [24] and SNQL [61]), for example. More general graph creation is also a feature of languages that adopt a functional or algebraic approach such as UnQL [13], GraphQL [37], and GOOD [34].

One approach to supporting node creation, followed in a number of languages, is to adopt a mechanism equivalent to that of Skolem functions, first used for semistructured data in the Mediator Specification Language MSL [54]. In a query language supporting recursion, it is important to ensure that, where possible, node creation and recursion do not combine to yield non-termination.

Recall that BiQL [24] can create new graphs from existing graphs. To do this new object identifiers are needed. They define semantics of their basic language in terms of a translation to Datalog, and then extend this by means of a single Skolem function to represent new object identifiers.

As mentioned in Section 2, SNQL [61] also uses Skolem functions to represent new nodes in the output of a query. Assume that we have a network representing people and the cities in which they live. City names are represented as values of the *livesIn* attribute associated with people. We want to transform this network into one in which cities become actors (rather than values), with *name* and *population* attributes. Although SNQL provides a graph-based syntax, the following Datalog-like program shows a simplification of how such a query might be translated

$$\begin{array}{lcl} ans(f(c), isa, city) & \leftarrow & temp(p,c) \\ ans(f(c), name, c) & \leftarrow & temp(p,c) \\ ans(f(c), population, count(p)) & \leftarrow & temp(p,c) \\ temp(p,c) & \leftarrow & (p, isa, person), (p, livesIn, c) \end{array}$$

Here, f is a Skolem function and *count* is an aggregate function. In general, more than one Skolem function may be needed in an SNQL query.

3.6 Approximate matching and ranking

In many applications, users may not be familiar with the graph structure being queried, its constraints or edge labels. As a result, they may formulate queries which return no answers or fewer answers than they expected. Early work to address such problems when querying semistructured data was done by Kanza and Sagiv [42], who proposed a form of flexible querying based on a notion of homeomorphism between a query and a graph.

In this section, we will consider a more general notion of approximate matching of paths [30, 40], where the results can be ranked in terms of their "closeness" to the original query. Consider a regular path query Q of the form given in (3), using regular expression r. Approximate matching is achieved by applying *edit operations* to $L(r)$. Possible edit operations include insertion, deletion, substitution, transposition and inversion of symbols. For simplicity, we will only consider the first three of these here.

Let $x, y \in \Sigma^*$. Applying an edit operation to x yielding y can be modelled as a binary relation $\rightsquigarrow$ over Σ^* such that $x \rightsquigarrow y$ holds iff there exist $u, v \in \Sigma^*$, $a, b \in \Sigma$, with $a \neq b$, such that one of the following is satisfied:

$$\begin{array}{lll} x = uav, & y = ubv & \text{(substitution)} \\ x = uav, & y = uv & \text{(deletion)} \\ x = uv, & y = ubv & \text{(insertion)} \end{array}$$

Let $\overset{k}{\rightsquigarrow}$ stand for the composition of $\rightsquigarrow$ with itself k times. The *edit distance* $d_e(x, y)$ between x and y is the minimum number k of edit operations such that $x \overset{k}{\rightsquigarrow} y$.

Each operation may have a different cost. In general, different instances of the same operation may have different costs. For example, a user may be prepared to substitute *taxi* by *train* at a cost of one, but *taxi* by *bus* at a cost of two.

More generally, Grahne and Thomo study approximate matching of RPQs in [30], where they

assume that approximations are specified by means of a *weighted regular transducer*. Such a transducer can be represented by a regular expression defined over triples of the form (a, k, b), which specify that a can be replaced by b with cost k.

A *weighted transducer* $T = (S_T, \Sigma, \sigma_T, S_{0_T}, F_T)$ comprises a finite set of states S_T, an input/output alphabet Σ, a set of initial states S_{0_T}, a set of final states F_T, and a transition relation $\sigma_T \subseteq S_T \times \Sigma \times \mathbb{N} \times \Sigma \times S_T$. A transition (s, a, k, b, t) specifies that if the transducer is in state s and reads symbol a, it outputs symbol b at cost k and moves to state t.

As stated in Section 3.2, we can construct an NFA M_r from the regular expression r in query Q and can consider graph G as an NFA as well. Now we can form the product automaton $M_r \times T \times G$ (see [30] for details). Then $(a, b, k) \in ans_T(Q, G)$ iff the shortest path from an initial state $(\_, \_, a)$ to a final state $(\_, \_, b)$ in $M_r \times T \times G$ has cost k. As noted in [30], if we are interested in nodes reachable from a limited number of source nodes, we can use Dijkstra's shortest path algorithm to return answers in increasing order of cost. Furthermore, we can construct the product automaton incrementally.

The simpler setting in which approximation is captured by edit operations, all with cost one, can be captured by a transducer T with a single state s and transitions from s to s labelled:

- $(a, 1, \varepsilon)$, for each $a \in \Sigma$ (deletion),
- $(\varepsilon, 1, a)$, for each $a \in \Sigma$ (insertion), and
- $(a, 1, b)$, for $a, b \in \Sigma$, $a \neq b$ (substitution).

Alternatively, in [40] ideas from approximate string matching [65] can be used to construct an *approximate* NFA M_Q from Q, from which the product of M_Q and M_G can be traversed. Extending approximate matching from RPQs to CRPQs, as well as adding an inversion edit operation for reversing the traversal of a graph edge, was studied in [40, 59]. Both the approximate NFA and the product automaton can be constructed incrementally, while any rank-join algorithm can be used on the conjuncts in order to return answers in increasing overall distance from the original CRPQ. This results in an algorithm with PTIME combined complexity if the conjuncts are acyclic and there is a fixed number of head variables [40].

4. EXPRESSIVE POWER AND COMPUTATIONAL COMPLEXITY

We now consider some results on the expressive power of graph query languages and the complexity of the QEP for such languages. For simplicity of notation, let us denote the classes of queries expressible by conjunctive queries, regular path queries, conjunctive regular path queries and extended conjunctive regular path queries by CQ, RPQ, CRPQ and ECRPQ, respectively. Furthermore, let FO denote the class of queries expressible in first-order logic (relational calculus or algebra). Then we have

$$\text{CQ} \subset \text{FO}$$

and

$$\text{RPQ} \subset \text{CRPQ} \subset \text{ECRPQ}.$$

For relating these latter classes with FO and with Datalog, we need some further definitions.

The language of first-order logic with transitive closure, denoted FO + TC (introduced by Immerman in [41]) extends first-order logic with formulas of the form $TC(\lambda\bar{x}, \bar{y}.\phi(\bar{x}, \bar{y}))$, where $\bar{x}$ are $\bar{y}$ are k-tuples, and $\phi(\bar{x}, \bar{y})$ is a formula in FO+TC denoting a binary relation on k-tuples. Then $TC(\lambda\bar{x}, \bar{y}.\phi(\bar{x}, \bar{y}))$ denotes the transitive closure of ϕ.

A *linear* Datalog program is one in which each rule has at most one recursive subgoal. A *stratified* Datalog program is one in which use of negated predicates is stratified. Let SL-DATALOG and GRAPHLOG denote the sets of queries expressible as stratified linear Datalog programs and in the language GraphLog (without aggregation), respectively. Then we have the following result [18]:

$$\text{FO} + \text{TC} = \text{GRAPHLOG} = \text{SL-DATALOG}$$

Similar expressive power is exhibited by STRUQL and UnQL. A theorem from [26] states that the closure of STRUQL under composition expresses precisely the Boolean queries expressible in FO + TC, while a theorem from [13] shows that all UnCAL queries can be expressed in FO+TC (where UnCAL is the algebra associated with UnQL). Now Immerman's result tells us that all GraphLog, STRUQL and UnQL queries can be computed in NLOGSPACE.

Adding aggregation operators to GraphLog in the form of closed semirings leaves query evaluation in NLOGSPACE, as long as the summarisation operators are in $\{min, max, +\}$ and aggregation operators are in $\{min, max\}$. However, when summarisation can include $\times$ and aggregation includes $+$, as needed to express the *parts explosion* query for example, then query evaluation is in NC^2 [20]

Further study of the expressiveness of stratified aggregation in Datalog was undertaken in [53]. They show that (1) Datalog extended with stratified negation cannot express a query to count the number of paths between every pair of nodes in an acyclic graph, (2) Datalog extended with stratified negation and arithmetic on integers (the + operator) can express all computable queries on ordered domains,

and (3) Datalog extended with stratified negation and generic function symbols can express all computable queries (on ordered and unordered domains).

Recently, Fletcher et al. [27] considered the relative expressive power of navigational graph query languages built from the operators identity, union, composition, intersection, set difference, projection, co-projection (values x such that there does not exist a y such that (x, y) is in the result of some expression), converse (i.e., inverse), transitive closure, and the diversity relation (all pairs of unequal constants in the active domain). They provide a complete comparison in terms of expressive power, both for path queries and Boolean queries.

The GOOD query language provides for greater expressive power than the other languages considered above. In [33, 34], it is shown that GOOD restricted to node addition/deletion and edge addition/deletion is relationally complete. Adding abstraction gives the expressive power of the nested relational algebra, while the full language including methods is Turing-complete.

The invention of values or object identifiers (oids) also adds power, as shown in database query languages such as IQL [1] and ILOG [14]. By relying on rules that use both recursion and oid invention, it can be shown that such languages can express all computable database queries [14]. However, when recursion and invention are not allowed to interact, queries can be evaluated in PTIME [1].

5. CONCLUSION

We have provided a survey of query languages for graph databases, focussing on a number of important aspects of functionality. While many language features/constructs have been the subject of formal study, work remains to be done in terms of an integrated and consistent framework in which to study graph query languages. In particular, the areas of approximate querying and graph transformations merit greater study.

The requirements of social network modelling and analysis provide further opportunities to extend the capabilities of graph languages. For example, many aspects of social network analysis rely on some probabilistic interpretation of graphs, so query languages need to be adapted and studied accordingly. Work in the area of expressive query languages for probabilitic databases has recently been initiated [23].

Acknowledgements

I would like to dedicate this survey to the memory of Alberto Mendelzon, whose insight and vision inspired me to investigate graph query languages some 25 years ago. I would also like to thank all the colleagues with whom I have collaborated.

6. REFERENCES

[1] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. *J. ACM*, 45(5):798–842, 1998.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The LOREL query language for semistructured data. *Int. J. Digit. Libr.*, 1(1):68–88, April 1997.

[3] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. Elsevier and MIT Press, 1990.

[4] B. Amann and M. Scholl. GRAM: A graph data model and query language. In *ECHT*, pages 201–211, 1992.

[5] S. Amer-Yahia, L. V. S. Lakshmanan, and C. Yu. SocialScope: Enabling information discovery on social content sites. In *CIDR*, 2009.

[6] M. K. Anand, S. Bowers, and B. Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, pages 287–298, 2010.

[7] R. Angles and C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.

[8] K. Anyanwu and A. P. Sheth. ρ-queries: enabling querying for semantic associations on the semantic web. In *WWW*, pages 690–699, 2003.

[9] M. Arenas, C. Gutiérrez, and J. Pérez. An extension of SPARQL for RDFS. In *SWDB-ODBIS*, pages 1–20, 2007.

[10] P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, pages 3–14, 2010.

[11] C. L. Barrett, R. Jacob, and M. V. Marathe. Formal-language-constrained path problems. *SIAM J. Comput.*, 30(3):809–837, 2000.

[12] P. Buneman. Semistructured data. In *PODS*, pages 117–121, 1997.

[13] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

[14] L. Cabibbo. The expressive power of stratified logic programs with value invention. *Inf. Comput.*, 147(1):22–56, 1998.

[15] D. Calvanese, G. de Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. *J. Comput. Syst. Sci.*, 64(3):443–465, 2002.

[16] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, pages 176–185, 2000.

[17] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *ACM Hypertext*, pages 269–292, 1989.

[18] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, pages 404–416, 1990.

[19] M. P. Consens and A. O. Mendelzon. Hy$^+$: a hygraph-based query and visualization system. In *SIGMOD*, pages 511–516, 1993.

[20] M. P. Consens and A. O. Mendelzon. Low complexity aggregation in graphlog and datalog. *Theor. Comput. Sci.*, 116(1&2):95–116, 1993.

[21] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *SIGMOD*, pages 323–330, May 1987.

[22] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. G$^+$:

Recursive queries without recursion. In *EDS*, pages 355–368, Redwood City, 1988. Benjamin/Cummings.
[23] D. Deutch, C. Koch, and T. Milo. On probabilistic fixpoint and markov chain query languages. In *PODS*, pages 215–226, 2010.
[24] A. Dries, S. Nijssen, and L. De Raedt. A query language for analyzing networks. In *CIKM*, pages 485–494, 2009.
[25] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.
[26] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with STRUDEL. *VLDB J.*, 9(1):38–55, 2000.
[27] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. V. den Bussche, D. V. Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs. In *ICDT*, pages 197–207, 2011.
[28] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, pages 139–148, 1998.
[29] D. D. Freydenberger and N. Schweikardt. Expressiveness and static analysis of extended conjunctive regular path queries. In *AMW*, 2011.
[30] G. Grahne and A. Thomo. Regular path queries under approximate semantics. *Ann. Math. Artif. Intell.*, 46(1-2):165–190, 2006.
[31] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
[32] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *VLDB*, pages 297–308, 1994.
[33] M. Gyssens, J. Paradaens, and D. V. Gucht. A graph-oriented object database model. In *PODS*, pages 417–424, 1990.
[34] M. Gyssens, J. Paredaens, J. V. den Bussche, and D. V. Gucht. A graph-oriented object database model. *IEEE TKDE*, 6(4):572–586, August 1994.
[35] D. Harel. On visual formalisms. *C. ACM*, 31(5):514–530, May 1988.
[36] S. Harris and A. Seaborne, editors. *SPARQL 1.1 Query Language*, W3C Working Draft, 12 May 2011.
[37] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *SIGMOD*, pages 405–418, 2008.
[38] J. Hidders. Typing graph-manipulation operations. In *ICDT*, pages 391–406, 2003.
[39] D. A. Holland, U. Braun, D. Maclean, K.-K. Muniswamy-Reddy, and M. I. Seltzer. Choosing a data model and query language for provenance. In *Proc. Int. Provenance and Annotation Workshop*, 2008.
[40] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Ranking approximate answers to semantic web queries. In *ESWC*, pages 263–277, 2009.
[41] N. Immerman. Languages that capture complexity classes. *SIAM J. Comput.*, 16(4):760–778, 1987.
[42] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *PODS*, pages 40–51, 2001.
[43] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.
[44] G. Klyne and J. J. Carroll, editors. *Resource Description Framework (RDF): Concepts and Abstract Syntax*, W3C Recommendation, 10 February 2004.
[45] K. Kochut and M. Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In *ESWC*, pages 145–159, 2007.
[46] Z. Lacroix, H. Murthy, F. Naumann, and L. Raschid. Links and paths through life sciences data sources. In *DILS*, pages 203–211, 2004.
[47] W.-J. Lee, L. Raschid, P. Srinivasan, N. Shah, D. L. Rubin, and N. F. Noy. Using annotations from controlled vocabularies to find meaningful associations. In *DILS*, pages 247–263, 2007.
[48] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):33–39, 2005.
[49] L. Libkin and Vrgoč. Regular path queries on graphs with data. In *ICDT (to appear)*, 2012.
[50] Y. A. Liu and S. D. Stoller. Querying complex graphs. In *PADL*, pages 199–214, 2006.
[51] F. Mandreoli, R. Martoglia, G. Villani, and W. Penzo. Flexible query answering on graph-modeled data. In *EDBT*, pages 216–227, 2009.
[52] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, December 1995.
[53] I. S. Mumick and O. Shmueli. How expressive is stratified aggregation? *Ann. Math. Artif. Intell.*, 15(3-4):407–434, 1995.
[54] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *VLDB*, pages 413–424, 1996.
[55] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260, 1995.
[56] J. Paredaens, P. Peelman, and L. Tanca. G-Log: A graph-based query language. *IEEE TKDE*, 7(3):436–453, 1995.
[57] J. Pérez, M. Arenas, and C. Gutiérrez. nSPARQL: A navigational language for RDF. In *ISWC*, pages 66–81, 2008.
[58] A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM TOIS*, 12(1):35–68, January 1994.
[59] A. Poulovassilis and P. T. Wood. Combining approximation and relaxation in semantic web path queries. In *ISWC*, pages 631–646, 2010.
[60] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE*, pages 1595–1602, 2009.
[61] M. San Martín, C. Gutiérrez, and P. T. Wood. SNQL: A social networks query and transformation language. In *AMW*, 2011.
[62] A. Sheth, B. Aleman-Meza1, I. B. Arpinar, C. Bertram, Y. Warke, C. Ramakrishnan, C. Halaschek, K. Anyanwu, D. Avant, F. S. Arpinar, and K. Kochut. Semantic association identification and knowledge discovery for national security applications. *J. Database Management*, 16(1):33–53, 2005.
[63] F. W. Tompa. A data model for flexible hypertext database systems. *ACM TODS*, 7(1):85–100, January 1989.
[64] G. Weikum, G. Kasneci, M. Ramanath, and F. M. Suchanek. Database and information-retrieval methods for knowledge discovery. *C. ACM*, 52(4):56–64, 2009.
[65] S. Wu and U. Manber. Fast text searching allowing errors. *C. ACM*, 35(10):83–91, Oct. 1992.