Expression and Efficient Processing of Fuzzy Queries in a Graph Database Context

Olivier Pivert, Grégory Smits, Virginie Thion

Expression and Efficient Processing of Fuzzy Queries in a Graph Database Context

Olivier Pivert, Grégory Smits, Virginie Thion
Rennes 1 University / IRISA
Lannion, France
Email: {Olivier.Pivert,Gregory.Smits,Virginie.Thion}@irisa.fr

***Abstract*—Graph databases have aroused a large interest in the last years thanks to their large scope of potential applications (e.g. social networks, biomedical networks, data stemming from the web). In a similar way as what has already been proposed in relational databases, defining a language allowing a flexible querying of graph databases may greatly improve usability of data. This paper focuses on the notion of fuzzy graph database and describes a fuzzy query language that makes it possible to handle such database, which may be fuzzy or not, in a flexible way. This language, called FUDGE, is an extension of the CYPHER language used for querying graph databases in a crisp way in the Neo4j graph database management system. It can be used to express preference queries on fuzzy graph databases. The preferences concern i) the content of the vertices of the graph and ii) the structure of the graph. The FUDGE language is implemented in a system called SUGAR, that we present in this article. We also discuss some implementation issues of the FUDGE language in SUGAR.**

INTRODUCTION

Much work has been done about fuzzy querying of *relational* databases, cf. for instance [21] or [31], which led in particular to a fuzzy extension of the SQL language, called SQLf [13]. However, even though relational databases are still widely used, the need to handle *complex* data has led to the emergence of other types of data models. In the last few years, a new concept has started to attract a lot of attention in the database world, namely that of *graph databases* (see e.g. [9], [35], [7]). The basic purpose of graph DB is to efficiently manage networks of entities where each node is described by a set of characteristics (e.g. a set of attributes), and each edge represents a link between entities. Such a database model has many potential applications, e.g. for modeling social networks, RDF data, cartographic databases, bibliographic databases, etc. Such a model may be extended into the notion of a fuzzy graph database where a degree may be attached to edges in order to express the "intensity" of any kind of gradual relationship (e.g., *likes*, *is friends with*, *is about*). Graph databases, which may be fuzzy or not, raise new challenges in terms of flexible querying since two aspects may be involved in the preferences that a user may express: i) the content of the nodes and ii) the structure of the graph.

In this paper, we present the FUDGE language and the SUGAR system implementing this language, that make it possible to query a graph database in a flexible way. The paper is organized as follows. In section I, we present some background notions about graph databases, fuzzy set theory, fuzzy graphs and the fuzzy querying of graph database. In Section II, we recall the basis of the query algebra underlying the FUDGE language, driven by a running example, which is also used to present the FUDGE language in the Section III. We then present the SUGAR system in Section IV, and focus on implementation issues. Related work is discussed in Section V. Section VI recalls the contributions and outlines some perspectives.

I. BACKGROUND NOTIONS

A. Graph databases

A graph database management system enables managing data for which the structure of the schema is modeled as a graph (nodes are entities and edges are relations between entities), and data is handled through graph-oriented operations and type constructors [9]. Among the existing systems, let us mention AllegroGraph [1], InfiniteGraph [2], Neo4j [3] and Sparksee [5]. There are different models for graph databases (see [9] for an overview), including the *attributed graph* (aka. *property graph*) aimed to model a network of entities with embedded data. In this model, nodes and edges may contain data in *attributes* (aka. *properties*).

B. Fuzzy graphs

A *graph* is a pair $(V,\ R)$, where V is a set and R is a relation on V. The elements of V (resp. R) correspond to the vertices (resp. edges) of the graph. Similarly, any fuzzy relation ρ on a set V can be regarded as defining a weighted graph, or fuzzy graph [33], [29], where the edge $(x,\ y) \in V \times V$ has weight or strength $\rho(x,\ y) \in [0,\ 1]$. As noted in [36], the fuzzy relation ρ may be viewed as a fuzzy subset on $V \times V$, which allows us to use much of the formalism of fuzzy sets. For example, we can say that $\rho_1 \subseteq \rho_2$ if $\forall(x,\ y),\ \rho_1(x,\ y) \leq \rho_2(x,\ y)$. Some notable properties that can be associated with fuzzy relations are reflexivity ($\rho(x,\ x) = 1,\ \forall x$), symmetry ($\rho(x,\ y) = \rho(y,\ x)$), transitivity ($\rho(x,\ z) \geq \max_y \min(\rho(x,\ y),\ \rho(y,\ z))$).

An important operation on fuzzy relations is composition. Assume ρ_1 and ρ_2 are two fuzzy relations on V. Thus, composition $\rho = \rho_1 \circ \rho_2$ is also a fuzzy relation on V s.t. $\rho(x,\ z) = \max_y \min(\rho_1(x,\ y),\ \rho_2(y,\ z))$. The composition operation can be shown to be associative: $(\rho_1 \circ \rho_2) \circ \rho_3 = \rho_1 \circ (\rho_2 \circ \rho_3)$. The associativity property allows us to use the notation $\rho^k = \rho \circ \rho \circ \ldots \circ \rho$ for the composition of ρ with itself $k-1$ times. In addition, following [36], we define ρ^0 to be s. t. $\rho^0(x,\ y) = 0,\ \forall(x,\ y)$.

Remark 1: Fuzzy graphs as defined above may be generalized to the case where a fuzzy set of vertices is considered.

Then, denoting by F the fuzzy subset of V considered, the corresponding fuzzy graph is defined as $(V,\ F,\ \rho_F)$. In this case, we let ρ_F be a relation on V defined as $\rho_F(x,\ y) = \min(\rho(x,\ y),\ \mu_F(x),\ \mu_F(y))$ where μ_F denotes the membership function attached to F. In the following, we only consider the simple case of a crisp set of vertices.

If ρ is symmetric, we shall say that $(V,\ \rho)$ is an undirected graph. Otherwise, we shall refer to $(V,\ \rho)$ as a directed graph. Without loss of generality, we consider directed graphs in the following.

C. Fuzzy preferences on graph DBs

In this section, we describe the main elements that may appear in a fuzzy query addressed to a graph database. Two types of preferences have to be considered: those on content and those on structure.

1) Preferences on the node content: The idea is to express flexible conditions about attributes associated with nodes and/or vertices of the graph. An example is: "find the people who are *young*, *highly educated*, and live in *Eastern Europe*" (assuming that each node contains information about the age, education level, address, etc., of the person it corresponds to). Compound conditions may also be expressed using a large range of fuzzy connectives. We do not get into more detail as this aspect has been much studied in a relational context [31].

2) Preferences on the graph structure : Hereafter, we describe the concepts of fuzzy graph theory that appear the most useful in a perspective of graph database querying. We denote a *fuzzy graph* by $G = (V,\ \rho)$.

Strength of a path. — A path p in G is a sequence $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$ $(n \geq 0)$ s.t. $\rho(x_{i-1},\ x_i) > 0,\ 1 \leq i \leq n$ and where n is the number of links in the path. The *strength* of the path is defined as

$$ST(p) = \min_{i=1..n} \rho(x_{i-1},\ x_i). \tag{1}$$

In other words, the strength of a path is defined to be the weight of the weakest edge of the path. Two nodes for which there exists a path p with $ST(p) > 0$ between them are called *connected*. We call p a cycle if $n \geq 2$ and $x_0 = x_n$. It is possible to show that $\rho^k(x,\ y)$ is the strength of the strongest path from x to y containing at most k links. Thus, the strength of the strongest path joining any two vertices x and y (using any number of links) may be denoted by $\rho^\infty(x,\ y)$.

Length and *distance*. — The *length* of a path $p = x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$ in the sense of ρ is a concept defined in [33] as:

$$Length(p) = \sum_{i=1}^{n} \frac{1}{\rho(x_{i-1},\ x_i)}. \tag{2}$$

Clearly $Length(p) \geq n$ (it is equal to n if ρ is Boolean, i.e., if G is a nonfuzzy graph). We can then define the *distance* between two nodes x and y in G as

$$\delta(x,\ y) = \min_{all\ paths\ x\ to\ y} Length(p). \tag{3}$$

It is the length of the shortest path from x to y. It can be shown that δ is a metric [33].

α-*cut of a relation*. — It is defined as: $\rho^\alpha = \{(x,\ y) | \rho(x,\ y) \geq \alpha\}$ where $\alpha \in]0,\ 1]$. Note that ρ^α is a crisp relation.

3) Preference combination: Different types of connectives may be considered for combining conditions about the content or the structure of the graph: "flat" (min, max, arithmetic mean, etc.), weighted (weighted mean, OWA, quantified proposition, etc, see [24]), or hierarchical.

II. Fuzzy graph databases and fuzzy algebra

In this section, we recall the main elements of the algebra that constitutes the foundations of the FUDGE language. The whole algebra is presented in detail in [32].

A. Data model

We are interested in fuzzy graph databases where nodes and edges can carry data (e.g. key-value pairs in attributed graphs, see Section I-A). So, we first propose an extension of the definition of a *fuzzy graph* into that of a *fuzzy data graph*.

Definition 1 (Fuzzy data graph): Let E be a set of labels. A *fuzzy data graph* G is a quadruple $(V,\ R,\ \kappa,\ \zeta)$, where V is a finite set of nodes (each node n is identified by $n.id$), $R = \bigcup_{e \in E} \{\rho_e : V \times V \rightarrow [0,\ 1]\}$ is a set of labeled fuzzy edges between nodes of V, and κ (resp. ζ) is a function assigning a (possibly structured) value to nodes (resp. edges) of G.

In the following, a *graph database* is meant to be a fuzzy data graph. The following example illustrates this notion.

Example 1: Fig. 1 is an example of a fuzzy data graph inspired from DBLP[1], with some fuzzy edges (with a degree in brackets), and crisp ones (degree equal to `1`). In this example, the degree associated with `A -contributor-> B` is the proportion of journal papers co-written by `A` and `B`, over the total number of journal papers written by `B`. Here, the degree is based on a simple statistical notion, but it could be made more sophisticated by the integration of expert knowledge. ⋄

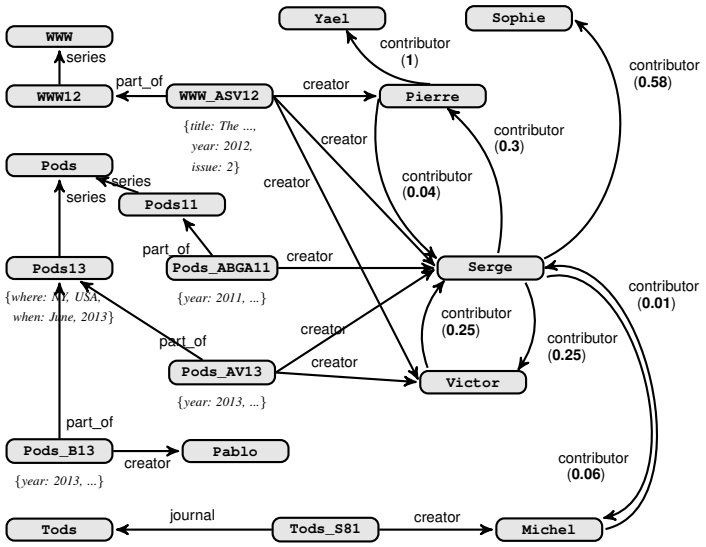

Fig. 1: A fuzzy data graph $\mathcal{DB}$ inspired of an excerpt of DBLP data

Nodes are assumed to be typed: if n is a node of V, then $Type(n)$ denotes its type. In $\mathcal{DB}$ (Fig 1), the nodes

[1] http://www.informatik.uni-trier.de/~ley/db/

WWW12, Pods11 and Pods13 are of type *Conference*, the nodes Pods_ABGA11, Pods_AV13, Pods_B13, Tods_S81, and WWW_ASV12 are of type *Article*, the nodes Pods, Tods, and WWW are of type *Series* and the other nodes are of type *Author*.

B. Algebra

We now move to the definition of a graph algebra suited to the definition of flexible queries. This algebra constitutes the core of the user-oriented language called FUDGE presented in section III. The whole algebra is described in [32]. Here, we focus on the main operator, namely *selection*, that handles fuzzy preferences. The basic unit of information is the graph.

The *selection* operator is based on the concept of *fuzzy graph pattern*, an extension of the crisp *graph pattern* notion [23] shown to have good properties for a practical implementation. We first introduce the notion of a *fuzzy regular expression*.

Definition 2 (Fuzzy regular expression): A *fuzzy regular expression* is an expression of the form

$$F ::= e \mid F{\cdot}F \mid F \cup F \mid F^* \mid F^{Cond}$$

where

- $e \in E \cup \{\_\}$ denotes an edge labeled by e, with the wildcard symbol denoting any label in E;
- $F{\cdot}F$ denotes a concatenation of expressions;
- $F \cup F$ denotes alternative expressions;
- F^* denotes the repetition of an expression;
- F^{Cond} denotes paths p satisfying F and the condition $Cond$ where $Cond$ is a boolean combination of atomic formulas of the form: Prop IS Fterm where Prop is a property defined on p and Fterm denotes a predefined or user-defined fuzzy term like short; see Fig. 2 (resp. Fig. 3), which gives a membership function associated with the fuzzy term *short* (resp. *recent*).

In the following, we limit properties to $\{\texttt{ST}, \texttt{Length}\}$ denoting resp. $ST(p)$ (See Eq. 1) and $Length(p)$ (See Eq. 2). Examples of conditions of this form are Length IS short and ST IS strong. Notice that Boolean conditions of the form Prop op a where a is a constant and op is a crisp comparator are a just special case.

In the following, giving a fuzzy regular expression f, f^+ is a shortcut notation for $f{\cdot}f^*$, f^k stands for $f{\cdot}f{\cdot}\cdots{\cdot}f$ with k occurrences of f and $f^{n,m}$ is a shortcut for $\bigcup_{i=n}^{m} f^i$.

A fuzzy regular expression is said to be *simple* if it is of the form e where $e \in E \cup \{\_\}$ (it denotes a *single* edge).

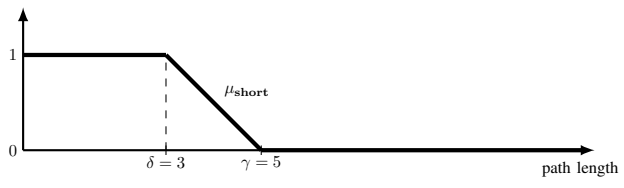

Fig. 2: Representation of the fuzzy term $short$

Definition 3 (Fuzzy regular expression matching): Given a path p and a fuzzy regular expression exp, p *matches* exp *with a satisfaction degree of* $\mu_{exp}(p)$ defined as follows, according to the form of exp (in the following, f, f_1 and f_2 are fuzzy regular expressions):

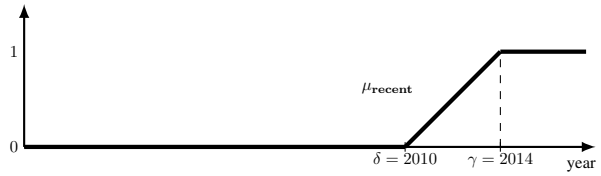

Fig. 3: Representation of the fuzzy term $recent$

- exp is of the form e with $e \in E$ (resp. "_"). If p is of the form $v_1 \xrightarrow{e'} v_1'$ where $e' = e$ (resp. where $e' \in E$) then $\mu_{exp}(p) = 1$ else $\mu_{exp}(p) = 0$.
- exp is of the form $f_1{\cdot}f_2$. Let P be the set of all pairs of paths (p_1, p_2) s.t. p is of the form p_1p_2. One has: $\mu_{exp}(p) = \max_P(min(\mu_{f_1}(p_1), \mu_{f_2}(p_2)))$.
- exp is of the form $f_1 \cup f_2$. One has: $\mu_{exp}(p) = \max(\mu_{f_1}(p), \mu_{f_2}(p))$.
- exp is of the form f^*. If p is an empty path then $\mu_{exp}(p) = 1$. Otherwise, we denote by P the set of all tuples of paths $(p_1, \cdots, p_n)$ $(n > 0)$ s.t. p is of the form $p_1 \cdots p_n$. One has: $\mu_{exp}(p) = \max_P(\min_{i\in[1..n]}(\mu_f(p_i)))$.
- exp is of the form f^{Cond} where $Cond$ is a (possibly compound) fuzzy condition. One has: $\mu_{exp}(p) = \min(\mu_f(p), \mu_{Cond}(p))$ where $\mu_{Cond}(p)$ is the degree of satisfaction of $Cond$ by p.

Not matching is equivalent to matching with a degree 0.

Example 2: Fig. 4 represents some paths from the graph database depicted in Fig. 1 that somewhat match the following fuzzy regular expressions:

- $e_1 = creator \cdot contributor^+$ is a fuzzy regular expression. All paths p_i ($i \in [1..4]$) of Fig. 4 match e_1 with a satisfaction degree of $\mu_{e_1}(p_i) = 1$.
- $e_2 = (creator \cdot contributor^+)^{\texttt{ST>0.4}}$ is a fuzzy regular expression. Path p_4 is the only one of Fig. 4 that matches e_2 (as $ST(p_1) = 0.3$, $ST(p_2) = 0.3$, $ST(p_3) = 0.01$ and $ST(p_4) = 0.58$), with $\mu_{e_2}(p_4) = 1$.
- $e_3 = creator \cdot (contributor^+)^{\texttt{Length IS short}}$, where $short$ is the fuzzy term of Fig. 2, is a fuzzy regular expression. Paths p_1, p_2 and p_4 of Fig. 4 match e_3 with $\mu_{e_3}(p_1) = 0.83$ as $\mu_{short}(1/0.3) = 0.83$ (where $1/0.3$ is the length of path from Serge to Pierre), $\mu_{e_3}(p_2) = 0.67$ as $\mu_{short}(1/0.3 + 1) = 0.67$ (where $1/0.67$ is the length of the $short$ path from Serge to Yael) and $\mu_{e_3}(p_4) = 1$ as $\mu_{short}(1/0.58) = 1$. Path p_3 does not match e_3 as $\mu_{short}(1/0.01) = 0$. $\diamond$

We then introduce the notion of a *fuzzy graph pattern*, which is a directed crisp graph with conditions on nodes and edges, types on nodes, and where edges are labeled by fuzzy regular expressions that denote paths.

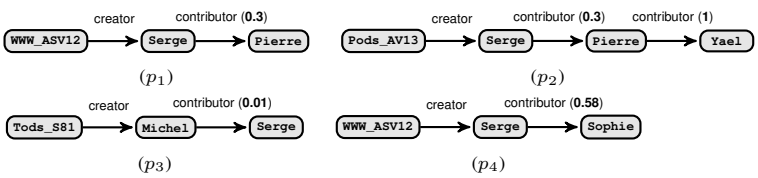

Fig. 4: Fuzzy regular expression matching

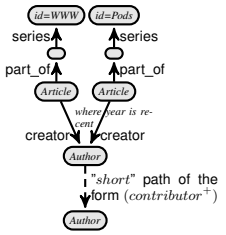

Fig. 5: Pattern $\mathcal{P}$

Definition 4 (Fuzzy graph pattern): Let $\mathcal{F}$ be a set of fuzzy terms. A *fuzzy graph pattern* is defined as a sextuple $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, f_e^{path}, f_n^{cond}, f_e^{cond}, f_n^{type})$ where

- $V_\mathcal{P}$ is a finite set of nodes;
- $E_\mathcal{P} \subseteq V_\mathcal{P} \times V_\mathcal{P}$ is a finite set of edges where (u, u') denotes an edge from u to u';
- f_e^{path} is a function defined on $E_\mathcal{P}$ s. t. for each (u, u') in $E_\mathcal{P}$, $f_e^{path}(u, u')$ is a fuzzy regular expression;
- f_n^{cond} is a function defined on $V_\mathcal{P}$ s. t. for each node u, $f_n^{cond}(u)$ is a condition on attributes of u, defined as a combination of atomic formulas of the form `A IS Fterm` where `A` denotes an attribute and `Fterm` denotes a fuzzy term (like e.g. `year IS recent`).
- f_e^{cond} is the counterpart of f_n^{cond} for edges. For each (u, u') in $E_\mathcal{P}$ for which $f_e^{path}(u, u')$ is simple, f_e^{cond} is the condition on attributes of (u, u'); and
- f_n^{type} is a function defined on $V_\mathcal{P}$ s. t. for each node u, $f_n^{type}(u)$ is the type of u.

In the following, we adopt a syntax *à la* CYPHER for graph pattern representation. CYPHER [30] is an intuitive query language inspired from ASCII-art for graph representation, implemented in the Neo4j (crisp) graph database management system [3]. A fuzzy graph pattern expressed *à la* CYPHER consists of a set of expressions `(n1:Type1)-[exp]->(n2:Type2)` or `(n1:Type1)-[e:label]->(n2:Type2)` where `n1` and `n2` are node variables, `e` is an edge variable, `label` is a label of E, `exp` is a fuzzy regular expression, and `Type1` and `Type2` are node types. Such an expression denotes a path satisfying a fuzzy regular expression exp (that is *simple* in the second form e) going from a node of type `Type1` to a node of type `Type2`. All its arguments are individually optional, so the merest form of an expression is `()-[]->()` denoting a path made of two nodes connected by any edge. Conditions on attributes are expressed on node and edges variables in a `WHERE` clause.

Example 3: We denote by $\mathcal{P}$ the fuzzy graph pattern:

```
1 (ar1:Article)-[part_of.series]->(s1),
2 (ar2:Article)-[part_of.series]->(s2),
3 (ar1)-[:creator]->(au1:Author),
4 (ar2)-[:creator]->(au1:Author),
5 (au1)-[(contributor+)|Length IS short]->(au2:Author)
6 WHERE
7 s1.id=WWW, s2.id=Pods,
8 ar2.year IS recent.
```

Listing 1: Pattern expressed *à la* CYPHER

Fig. 5 is a graphical representation of pattern $\mathcal{P}$ where the dashed edge denotes a path and information in italics denotes a node type or an additional condition on node or edge attributes. This pattern "models" information concerning authors (`au2`) who have, among their close contributors, an author (`au1`) who published a paper (`ar1`) in `WWW` and also published a paper (`ar2`) in `Pods` recently (`ar2.year IS recent`). ◇

Definition 5 (Fuzzy graph pattern matching): A (fuzzy) data graph $G = (V, R, \kappa, \zeta)$ *matches* a fuzzy graph pattern $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, f_e^{path}, f_n^{cond}, f_e^{cond}, f_n^{type})$ with a satisfaction degree denoted by $\mu_\mathcal{P}(G)$ if there exists a binary relation $S \subseteq V_\mathcal{P} \times V$ representing an injective function from $V_\mathcal{P}$ to V such that (i) for each node $u \in V_\mathcal{P}$, there exists a node $v \in V$ s. t. $(u, v) \in S$; (ii) for each edge $(u, u') \in E_\mathcal{P}$, there exist two nodes v and v' of V s. t. $\{(u, v), (u', v')\} \subseteq S$ and there is a path p in G from v to v' s. t. p matches $f_e^{path}(u, u')$ (recall that in case of matching, a satisfaction degree is associated, cf. Definition 3); (iii) for each pair $(u, v) \in S$, $\kappa(v) \vdash f_n^{cond}(u)$ (the semantics of $\vdash$ is clear from the context here) and $f_n^{type}(u) = Type(v)$ and (iv) the same reasoning is trivially applied to conditions on attributes for edges labeled with a simple fuzzy regular expression in $E_\mathcal{P}$, that is to say $\zeta(v, v') \vdash f_e^{cond}(u, u')$.

The value of $\mu_\mathcal{P}(G)$ is the minimum of the satisfaction degrees produced by the mappings and conditions from (ii), (iii) and (iv). If there is no relation S satisfying the previous conditions, then $\mu_\mathcal{P}(G) = 0$, i.e., G does not match $\mathcal{P}$.

Example 4: Fig. 6 gives the set of subgraphs of $\mathcal{DB}$ matching the pattern $\mathcal{P}$ of Example 3. Note for the following that $\mu_{recent}(2011) = 0.25$ and $\mu_{recent}(2013) = 0.75$. We note p the path going from $au1$ to $au2$. Let us now consider the satisfaction degree associated with each graph of Fig. 6. As the satisfaction degree is the minimum of the satisfaction degrees induced by lines 5 and 8, we have $\mu_\mathcal{P}(g_1) = 0.75$ (as $\mu_{short}(Length(p)) = \mu_{short}(1.72) = 1$ and $\mu_{recent}(2013) = 0.75$), $\mu_\mathcal{P}(g_2) = 0.5$ (as $\mu_{short}(Length(p)) = \mu_{short}(4) = 0.5$ and $\mu_{recent}(2013) = 0.75$), $\mu_\mathcal{P}(g_3) = 0.33$, (as $\mu_{short}(Length(p)) = \mu_{short}(4.33) = 0.33$ and $\mu_{recent}(2013) = 0.75$) $\mu_\mathcal{P}(g_4) = 0.75$, $\mu_\mathcal{P}(g_5) = 0.5$, (as $\mu_{short}(Length(p)) = \mu_{short}(4) = 0.5$ and $\mu_{recent}(2013) = 0.75$) $\mu_\mathcal{P}(g_6) = 0.25$ (as $\mu_{short}(Length(p)) = \mu_{short}(1.72) = 1$ and $\mu_{recent}(2011) = 0.25$), $\mu_\mathcal{P}(g_7) = 0.25$, $\mu_\mathcal{P}(g_8) = 0.25$, and $\mu_\mathcal{P}(g_9) = 0.4$.◇

Let us now move to the definition of the *selection* operator of the algebra. Even if the graph database contains a single graph, a query may return a set of graphs as several subgraphs may match a pattern as shown in Example 3. Graphs of a set do not necessarily have the same structure. A satisfaction degree is associated with each graph. A set of pairs $\langle\, graph\,, degree\,\rangle$ is nothing but a fuzzy set of graphs. Hence, each operator of the algebra takes one or more (depending on the arity of the

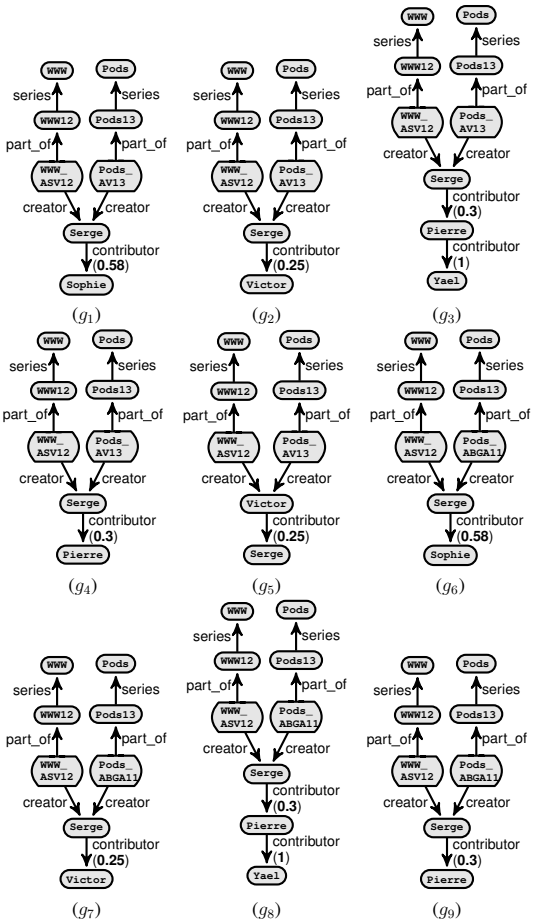

Fig. 6: Subgraphs of $\mathcal{DB}$ matching $\mathcal{P}$

operation) fuzzy set(s) of graphs as input and generates a fuzzy set of graphs as an output. All of the operators of the algebra operate in closed form. Applying an operation to the whole initial database means applying the operation to the singleton $\{\langle\, \mathcal{DB}\,,\, 1\,\rangle\}$. *Expressions of the algebra* are defined inductively as usual: (i) a (fuzzy) graph database $\mathcal{DB}$ is an expression of the algebra, and (ii) if $e_1, \cdots, e_n$ are expressions and O is an operator of arity n, then $O(e_1, \cdots, e_n)$ is an expression of the algebra.

Definition 6 (Selection operator): The selection operator σ takes as an input a fuzzy graph pattern $\mathcal{P}$ and a fuzzy set $\mathcal{G}$ of graphs. It returns a fuzzy set composed of all subgraphs of $\mathcal{G}$ that match the fuzzy graph pattern.

$$\sigma_{\mathcal{P}}(\mathcal{G}) = \{\langle\, s\,,\, \min(d,\, \mu_{\mathcal{P}}(s))\,\rangle \,|\, \mu_{\mathcal{P}}(s) > 0\}$$

where s is a subgraph of g such that $\langle\, g\,,\, d\,\rangle \in \mathcal{G}$. In case of duplicates (a same graph appearing with several satisfaction degrees), the highest satisfaction degree is kept.

III. The FUDGE language

The FUDGE language [32] is an extension of the CYPHER language [30], used for querying graph databases in a crisp way in the Neo4j graph DBMS [3]. It is based on the algebra defined in Section II. As for the algebra, we focus on the selection operation.

Given a graph database $\mathcal{DB}$, a selection query $\sigma_{\mathcal{P}}(\mathcal{DB})$ expressed in the FUDGE language is composed of:

1) a list of `DEFINE` clauses for fuzzy term declarations. If a fuzzy term `fterm` corresponds to a trapezoidal function with the four positions (abscissa) `A-a`, `A`, `B` and `B+b`, then the clause has the form `DEFINE fterm AS (A-a,A,B,B+b)`. If `fterm` is a decreasing function like the term $short$ of Fig. 2, then the clause has the form `DEFINEDESC fterm AS` (γ, δ) (there is the corresponding `DEFINEASC` clause for increasing functions, like the term $recent$ of Fig. 3).
2) a `MATCH` clause of the form `MATCH pattern WHERE conditions`, where `pattern` denotes a the fuzzy graph pattern $\mathcal{P}$.

Listing 2 is an example of a FUDGE query. The `DEFINEDESC` clause defines the fuzzy term $short$ of Fig. 2, and the next clause defines the fuzzy term $recent$. The pattern defined in lines 4 to 10 is the one of Example 3.

```
1  DEFINEDESC short AS (3,5)
2  DEFINEASC recent AS (2010,2014)
3  IN
4  MATCH
5    (ar1:Article)-[part_of]->()->[series]->(s1),
6    (ar2:Article)-[part_of]->()->[series]->(s2),
7    (ar1)-[:creator]->(au1:Author),
8    (ar2)-[:creator]->(au1:Author),
9    (au1)-[(contributor+)|Length IS short]->(au2:Author)
10 WHERE s1.id=WWW AND s2.id=Pods AND ar2.year IS recent
```

Listing 2: A FUDGE query

The FUDGE language is implemented in a system called SUGAR, presented in the next section.

IV. The SUGAR system

The SUGAR software is based on the Neo4j system [3] that implements the CYPHER (crisp) query language. SUGAR extends the interactive Neo4j REPL Console Rabbithole [4]. We discuss hereafter some implementation issues of SUGAR.

A. Modeling fuzzy graph databases in Neo4j

Neo4j is a management system for crisp property graph databases. In a crisp property graph, a set of properties (key-value pairs) can be bound to a node or an edge. Properties usually denote embedded data and meta-data for nodes, and properties of the relation for edges. We use a simple mechanism for simulating fuzzy graph databases in this crisp data model: we attach to each edge of the property graph a supplementary property called *fdegree* carrying the degree value of the relation, supposing that *fdegree* now becomes a reserved keyword of the system. If needed, a similar mechanism allows to turn crisp nodes into fuzzy ones.

B. FUDGE query evaluation

The SUGAR software is composed of two modules, which interact with the Neo4j crisp engine: the *Transcriptor module*, aimed to translate a FUDGE query into a (crisp) CYPHER one, which is then sent to the crisp Neo4j engine, and the *Score Calculator module*, which calculates the satisfaction degree associated with each answer returned by the crisp engine, and

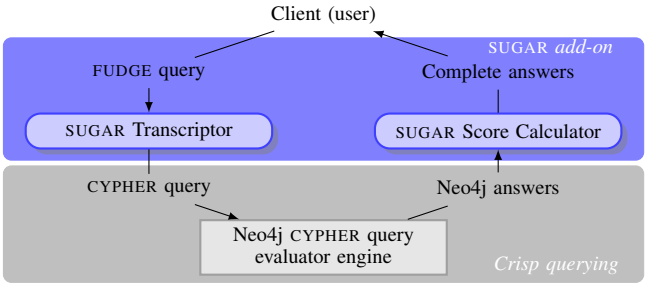

Fig. 7: SUGAR software architecture

the ranking of the answers. Fig. 7 illustrates this architecture. Let us now detail the three stages of a FUDGE query evaluation.

Stage 1 (Transcription) – The SUGAR Transcriptor module translates a FUDGE query into a CYPHER crisp one, allowing not only to retrieve the subgraphs that are isomorphic to the graph pattern of the FUDGE query, but also to retrieve information needed for the calculation of the satisfaction degree of each answer. The transcription implements i) the calculation of $Length$ and ST (see Section I-C2) for each concerned path, ii) the pattern matching without taking fuzzy preferences into account and iii) the retrieval of information needed for the next step of the evaluation.

As a first improvement step, the transcription implements a *derivation* evaluation method [14], which consists in reducing the set of candidate answers by translating the level of the α-cut in crisp conditions. As a simple illustration of this principle, the fuzzy condition `year IS recent` induces using the crisp condition `year > 2010` in order to remove the answers that necessary do not belong to the support of the answer.

Example 5: The crisp CYPHER query of Listing 3 is the transcription of the FUDGE query of Listing 2.

```
1  MATCH
2    (ar1:Article)-[:PART_OF]->()-[:SERIES]->(s1),
3    (ar2:Article)-[:PART_OF]->()-[:SERIES]->(s2),
4    (ar1)-[:CREATOR]->(au1:Author),
5    (ar2)-[:CREATOR]->(au1:Author),
6    p1 = (au1)-[CONTRIBUTOR*]->(au2:Author)
7  WITH
8  REDUCE(length=0, edge IN relationships(p1)|length
9       + 1/edge.fdegree) AS length_au1_au2_p1,
10      ar1 AS ar1, s1 AS s1, ar2 AS ar2, s2 AS s2,
11      au1 AS au1,au2 AS au2
12 WHERE s1.name='WWW' AND s2.name='Pods'
13      AND ar2.year>2010 AND length_au1_au2_p1<5.0
14 RETURN ar1, s1, ar2, s2, au1, au2,
15      length_au1_au2_p1 AS calc_fuzzy_length_p1_short,
16      ar2.year AS calc_fuzzy_ar2_year_recent
```

Listing 3: Crisp CYPHER query, after the transcription step

Lines 2 to 6 refer to the graph pattern structure. Lines 8 to 9 perform the calculation of the length of the path connecting `au1` to `au2`. The `WHERE` clause implements crisp conditions of the initial query (line 12) and conditions induced by the derivation of fuzzy preferences (line 13). The `RETURN` clause returns the isomorphic subgraphs that belong to the answer (line 14), and complementary information (lines 15 and 16) needed for the *Score Calculation* stage. Complementary information are the values of elements for which the grade of the membership in a fuzzy set has to be evaluated in order to obtain the satisfaction degree associated to the answer. For the query of Example 5, the information needed is the length of the path connecting `au1` to `au2` and the year of article `ar2`.◇

Stage 2 (Crisp evaluation of the translated query) – The translated query is sent to the crisp Neo4j query evaluation engine, that returns the result of its evaluation over the fuzzy graph database. It constitutes an intermediate result.

Example 6: The result of the evaluation of the crisp query of Listing 3 on the fuzzy graph database of Fig. 1 is:

Subgraph answer	Additional information `Length` of `p1` [1]	`year` of `ar2` [2]
g_1	1.72	2013
g_2	4.0	2013
g_3	4.33	2013
g_4	3.33	2013
g_5	4.0	2013
g_6	1.72	2011
g_7	4.0	2011
g_8	4.33	2011
g_9	3.33	2011

[1] needed for the evaluation of grade membership to *short*.
[2] needed for the evaluation of grade membership to *recent*.
where subgraphs $\{g_i | i \in [1..9]\}$ are those of Fig. 6).◇

Stage 3 (Score calculation and ranking of answers) – For each answer, the *Score Calculator* module evaluates the grade of the membership of each additional information to the adequate fuzzy set, and then deduces the aggregated satisfaction degree. The module also ranks the answers by decreasing order of satisfaction degree.

Example 7: Fig. 8 presents a screenshot of the SUGAR GUI, which contains the final result of the evaluation of the running example. The GUI is composed of two frames:

- a central frame for vizualizing the graph and the results of a query (a graphical representation of the graph is overprinted), and
- an input field frame (placed under the central one), for entering and running a FUDGE query.◇

C. The cost of flexibility

We discuss the additional cost, induced by the introduction of flexibility, for the selection query evaluation problem. We do not consider the other problem of considering a fuzzy graph, which intrinsically makes more complex some graph properties s.t. the *length* or the *distance* (see e.g. [12]). Let us consider the evaluation of a FUDGE query $\mathcal{Q}_{FUDGE}$, which includes z occurrences of fuzzy terms, over a graph database $\mathcal{G}$.

The first stage, the *transcription*, is computed in linear time of the size of the FUDGE query. A new query called $\mathcal{Q}_{CRISP}$ is produced. It contains at most $2z$ additional conditions in the `WHERE` clause, produced by the derivation mechanism. Note that a user, who would have wanted the same information (without satisfaction degree associated to the answers), should have written a crisp query equivalent to $\mathcal{Q}_{CRISP}$. We then cannot say that introducing fuzzy terms increases the size of the query.

The second stage is the *evaluation of the crisp query*. We denote by $\mathcal{A}$ the set of answers of $\mathcal{Q}_{CRISP}$ over $\mathcal{G}$. Computing

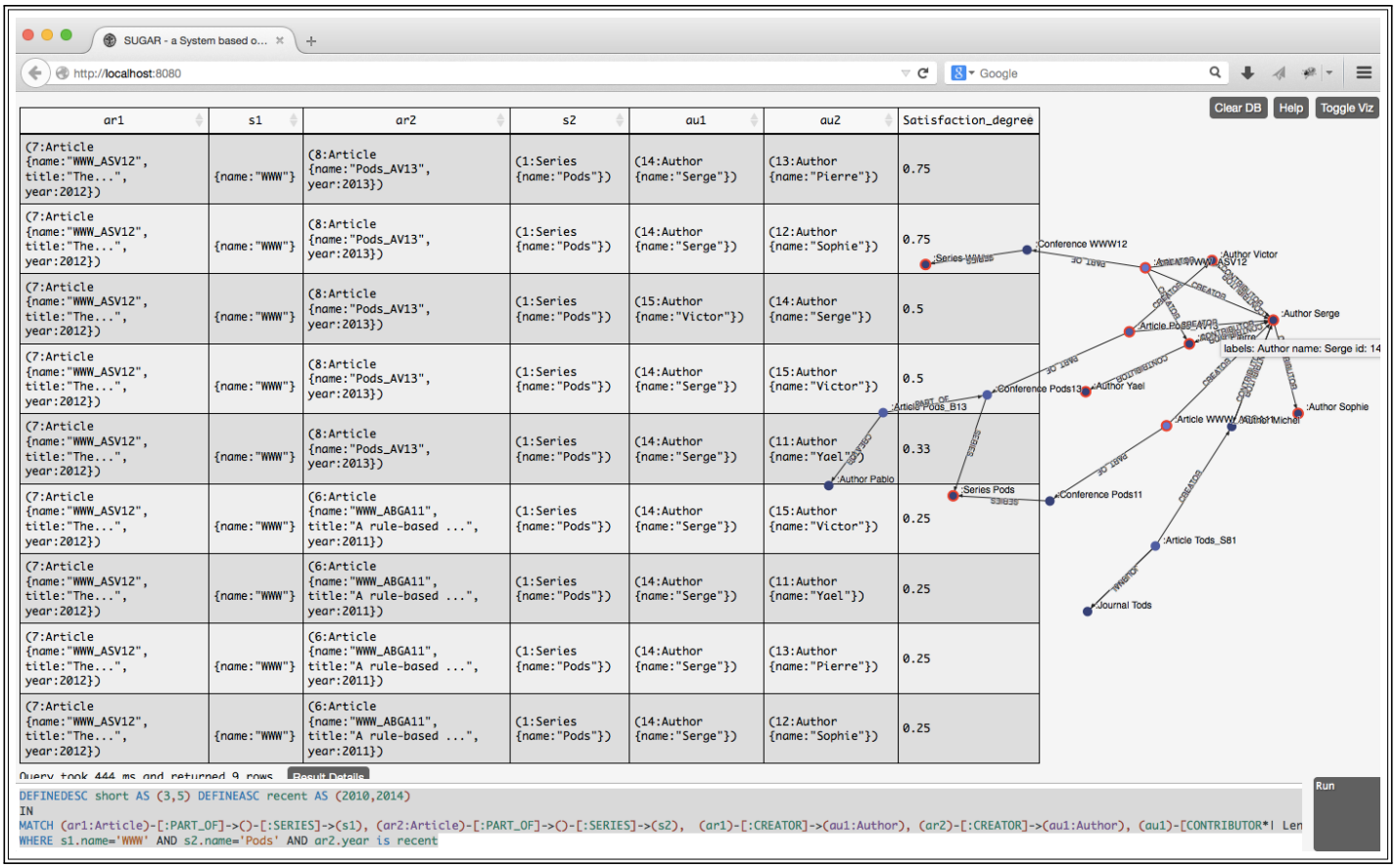

Fig. 8: Screenshot of the SUGAR system

$\mathcal{A}$ is a graph pattern isomorphism problem. Such problem was intensively studied in literature. In our case, we can bring back to the evaluation of graph pattern queries containing regular expressions. In [22] authors propose an algorithm for computing the result of such a query on a data graph $\mathcal{G}$ in $\mathcal{O}(|V|^3)$, where V is the set of vertices of $\mathcal{G}$.

The last stage, the *score calculation*, consists in (i) Computing the satisfaction degree of each answer. This is done in $\mathcal{O}(|\mathcal{A}| \times 2|z|)$ (ii) Ranking the set of answers according their satisfaction degree ($|\mathcal{A}| \times log(|\mathcal{A}|)$).

It seems obvious that first and third stages that permit to introduce flexibility in the query language are strongly dominated in complexity by the crisp evaluation (second stage).

V. Related work

As several models have been proposed to represent data having an implicit or explicit graph structure (see [9] for an overview), literature includes a variety of query languages for graphs. Authors of [9], [35] and [11] propose complementary surveys of graph query languages defined in the past 25 years, including languages for querying graph-based object databases, semi-structured data, social networks and semantic web data. Reference [11] focuses on theoretical query languages for graph databases, and emphasizes that graph database management systems still lack query languages with a clear syntax and semantics. Our work goes towards filling this gap.

Functionalities that should be offered by a language for querying the topology of a crisp graph database are exhibited in [8], [7], [34], [19], [10], [35]. We summarize these (non-exclusive) functionalities hereafter, focusing on selection statements. Given a graph data G, *adjacency queries* test node adjacency e.g. check whether two nodes are adjacent, list all neighbors of a node; Given a vertex, *reachability queries* search for topologically related vertices in G, where vertices are reachable by a *fixed-length path*, a *regular simple path* or a *shortest path*; *pattern matching queries* look for all subgraphs of G that are isomorphic to a given graph pattern; *data queries* specify conditions on the data embedded in G. Our algebra for fuzzy graph database expresses some flexible adjacency, reachability (as a rooted path is a special case of graph pattern), pattern matching and data queries on fuzzy and crisp graph databases. As a satisfaction degree is attached to each answer, rank-ordering them is straightforward.

Concerning flexible querying, [36] discusses different types of fuzzy preference criteria that appear relevant in the context of graph databases, without getting into the detail of how to express them using a formal query language. There are three main approaches allowing a flexible querying of graph databases: (i) *keyword-based query* approaches that completely ignore the data schema (see e.g. [25]), which lack expressiveness for most querying use cases [28]; (ii) approaches that, given a "crisp" query, propose *approximate answers*, for instance by implementing query relaxation or

an approximate matching mechanism (see e.g. [26], [15] or [28]); (iii) approaches allowing the user to introduce flexibility when formulating the query. Our approach belongs to this latter family for which many contributions concern the flexible extension of XPath [20], [16], [6] for querying semi-structured data (data trees). Such navigational languages behave well for querying graph databases in a crisp way [27] but no flexible extension was proposed in this specific case.

A work somewhat close to ours is [18] where authors propose a flexible extension of SPARQL allowing to introduce fuzzy terms and relations into the query language. A proof-of-concept implementation is proposed. This work only considers crisp graph databases, though.

Unlike other fuzzy extensions of CYPHER, e.g. [17], FUDGE relies on a formal algebra [32], whose expressiveness goes far beyond the simple examples presented here. To our knowledge, the FUDGE language and the SUGAR system are the only concrete contributions that consider *fuzzy* graph databases.

VI. Conclusion

In this paper, we have presented a language called FUDGE that makes it possible to query fuzzy graph databases in a flexible way (a crisp graph database being a special case of fuzzy graph database). The FUDGE language allows to express preferences on the data embedded in the graph and the structure of the graph (which may itself be fuzzy). We presented the SUGAR system which implement this language and discuss some implementation issues of interest, including the cost of introducing such a flexibility in selection graph pattern queries.

References

[1] AllegroGraph web site. franz.com/agraph/allegrograph.

[2] InfiniteGraph web site. www.objectivity.com/infinitegraph.

[3] Neo4j web site. www.neo4j.org.

[4] RabbitHole Console. http://neo4j.com/blog/rabbithole-the-neo4j-repl-console/.

[5] Sparksee web site. sparsity-technologies.com.

[6] J. M. Almendros-Jiménez, A. Luna, and G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In *Proc. of the RuleML*, pages 186–193. Springer-Verlag, 2011.

[7] R. Angles. A comparison of current graph database models. In *Proc. of ICDE Workshops*, pages 171–177, 2012.

[8] R. Angles and C. Gutiérrez. Querying RDF Data from a Graph Database Perspective. In *Proc. of European Semantic Web Conf. (ESWC)*, pages 346–360, 2005.

[9] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, 2008.

[10] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking database systems for social network applications. In *Proc. of the Intl. Workshop on Graph Data Management Experiences and Systems*, 2013.

[11] P. Barceló Baeza. Querying graph databases. In *Proc. of PODS*, pages 175–188, 2013.

[12] P. Bhattacharya and F. Suraweera. An algorithm to compute the supremum of max-min powers and a property of fuzzy graphs. *Pattern Recognition Letters*, 12(7):413–420, 1991.

[13] P. Bosc and O. Pivert. SQLf: a relational database language for fuzzy querying. *IEEE Trans. on Fuzzy Systems*, 3:1–17, 1995.

[14] P. Bosc and O. Pivert. *Knowledge Management in Fuzzy Databases*, chapter SQLf query functionality on top of a regular relational database management system. Heidelberg, Germany, 2000.

[15] P. Buche, J. Dibie-Barthélemy, and G. Hignette. Flexible Querying of Fuzzy RDF Annotations Using Fuzzy Conceptual Graphs. In *Proc. of ICCS*, pages 133–146, 2008.

[16] A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, and P. Spoletini. A Fuzzy Extension of the XPath Query Language. *J. Intell. Inf. Syst.*, 33(3):285–305, 2009.

[17] A. Castelltort and A. Laurent. Fuzzy queries over nosql graph databases: Perspectives for extending the cypher language. In *Proc. of IPMU*, 2014.

[18] J. Cheng, Z. M. Ma, and L. Yan. f-SPARQL: A flexible extension of SPARQL. In *Proc. of DEXA*, pages 487–494, 2010.

[19] M. Ciglan, A. Averbuch, and L. Hluchý. Benchmarking traversal operations over graph databases. In *Proc. of ICDE Workshops (ICDEW)*, pages 186–189, 2012.

[20] E. Damiani, S. Marrara, and G. Pasi. FuzzyXPath: Using Fuzzy Logic an IR Features to Approximately Query XML Documents. In *Found. of Fuzzy Logic and Soft Computing*, pages 199–208. Springer, 2007.

[21] D. Dubois and H. Prade. Using fuzzy sets in database systems: Why and how? In *Proc. of FQAS*, pages 89–103, 1996.

[22] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *Proc. of the ICDE*, pages 39–50, Washington, DC, USA, 2011. IEEE Computer Society.

[23] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, 6(3):313–338, 2012.

[24] J. Fodor and R. Yager. Fuzzy-set theoretic operators and quantifiers. In *The Handbooks of Fuzzy Sets Series, vol. 1: Fundamentals of Fuzzy Sets*, pages 125–193. Kluwer Academic Publishers, 2000.

[25] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. In *Proc. of SIGMOD*, pages 305–316, 2007.

[26] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proc. of PODS*, pages 40–51, 2001.

[27] L. Libkin, W. Martens, and D. Vrgoč. Querying Graph Databases with XPath. In *Proc. of ICDT*, pages 129–140, 2013.

[28] F. Mandreoli, R. Martoglia, G. Villani, and W. Penzo. Flexible query answering on graph-modeled data. In *Proc. of EDBT*, pages 216–227, 2009.

[29] J. N. Mordeson and P. S. Nair. *Fuzzy Graphs and Fuzzy Hypergraphs*, volume 46 of *Studies in Fuzziness and Soft Computing*. Springer, 2000.

[30] Neo Technology. The Neo4j Manual v2.0.0, 2013. Part III.

[31] O. Pivert and P. Bosc. *Fuzzy Preference Queries to Relational Databases*. Imperial College Press, 2012.

[32] O. Pivert, V. Thion, H. Jaudoin, and G. Smits. On a fuzzy algebra for querying graph databases. In *Proc. of IEEE ICTAI*, pages 748–755, 2014.

[33] A. Rosenfeld. Fuzzy graphs. In *Fuzzy Sets and their Applications to Cognitive and Decision Processes*, pages 77–97. Academic Press, 1975.

[34] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *ACM Southeast Regional Conf.*, page 42, 2010.

[35] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[36] R. Yager. Social network database querying based on computing with words. In *Flexible Approaches in Data, Information and Knowledge Management*, Studies in Computational Intelligence. Springer, 2013.