Functional querying in graph databases

Jaroslav Pokorný¹

Received: 12 July 2017 / Accepted: 16 October 2017 / Published online: 10 November 2017
© The Author(s) 2017. This article is an open access publication

Abstract The paper is focused on a functional querying in graph databases. We consider labelled property graph model and mention also the graph model behind XML databases. An attention is devoted to functional modelling of graph databases both at a conceptual and data level. The notions of graph conceptual schema and graph database schema are considered. The notion of a typed attribute is used as a basic structure both on the conceptual and database level. As a formal approach to declarative graph database querying a version of typed lambda calculus is used. This approach allows to use a logic necessary for querying, arithmetic as well as aggregation function. Another advantage is the ability to deal with relations and graphs in one integrated environment.

Keywords Graph database · Querying graph database · Graph database schema · Graph conceptual schema · Functional graph database schema · Functional graph database · Typed lambda calculus · Language of terms · XML graph

1 Introduction

Graph databases are focused on efficient storing and querying highly connected data. They are a powerful tool for graph-like queries, e.g., computing the shortest path between two nodes in the graph. They reach an excellent performance for local reads by traversing the graph and can use various data models for graphs and their data extensions.

Graph databases are considered usually as NoSQL databases (e.g., [18]). One rather popular definition of a *graph database* (GDB), also called a *graph-oriented database*, says that it is a database that uses graph theory to store, map and query relationships. That is, the distinguished characteristics of the domain include:

- relationship-rich data,
- relationships are first-class citizens in graph databases.

Despite of the fact that there are various approaches to GDB implementation, native graph processing based on so called *index-free adjacency* is the most efficient means of processing data in a graph because connected nodes use physical “pointers” to neighbour nodes in the database. Of course, there are other approaches based on an extension of SQL. For example, graph features introduced in SQL Server 2017¹ allow users to create node or edge tables. Graph extensions are fully integrated in SQL Server engine.

A GDB can contain a single (big) graph or a collections of graphs. The former includes, e.g., graphs of social platforms such as, Facebook, Twitter, Linked-In or Web graph, the latter is especially used in scientific domains such as bioinformatics and chemistry or human interaction patterns, temporal road networks, etc. Graph search occurs in other application scenarios, like recommender systems, complex software plagiarism detection, and traffic route planning. In line with similar concepts in other database technologies, we will talk about *graph data management systems* (GDBMS) and graph database systems (GDBS).

An important part of GDB technology is querying graphs. Always there is the intimate relationship between database

✉ Jaroslav Pokorný
pokorny@ksi.mff.cuni.cz

¹ MFF UK, Malostranské nám.25, 118 00 Prague, Czech Republic

¹ <https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-architecture> (retrieved on 10.9.2017).

modelling and querying. Most graph query languages use directly a structure of directed graphs or property graphs. Now, the most known declarative query language over property graphs is Cypher² of GDBMS Neo4j [15]. Cypher was the first pattern-matching query language to target the property graph data model. Cypher commands use partially SQL syntax and are targeted at ad hoc queries over the graph data.

Yet other approaches are possible, e.g., a functional approach. In the late 1980s, there was the functional language, DAPLEX [16]. The language only allowed nested applications of functions. The functional map was applied in context-oriented semantics of multivalued functions. A number of significant works using functional approach to data management are contained in the book [4]. In the current era of GDBMSs, we can mention the Gremlin³—a functional graph query language developed by Apache TinkerPop which allows to express complex graph traversals and mutation operations over property graphs. Traversal operators/functions are chained together to form path-like expressions. Gremlin is supported by many GDBMSs (e.g., Titan⁴).

Here, we will use a functional approach in which a database graph is represented by so called attributes, i.e., typed partial functions. We use for this approach the HIT Database Model, see, e.g., [7], as a functional alternative variant of E-R model. Then a typed lambda calculus, i.e., the language of lambda terms (LT), can be used as a data manipulation language.

Due to the strong typing, LT can deal with various data structures in a natural way as with functions. Since sets (relations) are modelled as their characteristic functions, we gain a tool for common manipulation of relations and graph data. In consequence, the query results can include new graphs, relations or nested relations as well. In practice, attempts to combine, e.g., Neo4j and Oracle DBMS already exist [17]. In this polyglot environment it is possible to synchronize a portion of the data from the relational database to Neo4j or even to synchronize all data between Oracle and Neo4j. To use a high-level formal apparatus like a variant of the LT language as a background for combining these two database technologies can be beneficial for practice.

We will see, that this functional approach reflects the graph structure of a GDB and, moreover, provides powerful possibilities for dealing with properties, i.e., with the GDB content. Attribute descriptions simultaneously provide a conceptual view of the data in the GDB.

In early 2000s we used the functional approach in the context of the XML language [8]. In a little simplified view,

XML⁵ data are trees. One can view XML data as functions from elements to PCDATA or to more complex data structures composed from additional elements and strings. An XML variant appropriate for querying XML data called XML- λ was described in [8]. Thus, it would be also possible to integrate querying over relational and XML data. Now it is possible in SQL, e.g., to work with XML data type in a table column (SQL/XML), but a unified query language for such polyglot databases is still missing.

The goal of the paper is to present above mentioned approaches and discuss their power and usability. We applied the functional approach to properties graphs in the work [14]. The present paper is an extension of [14].

The rest of the paper is organized as follows. Section 2 introduces a graph data model based on (labelled) property graphs. In addition, we will consider XML graphs as a special case. Section 3 describes modelling and querying GDBs, both on the conceptual and database level. XML graphs are also considered. The notions of graph conceptual schema and graph database schema are introduced including some integrity constraints (ICs). Section 4 shortly introduces a functional approach to GDB modelling based on typed attributes. A version of typed lambda calculus appropriate for GDB querying is introduced in Sect. 5. Details of querying GDB with functions are explained in examples including functional querying XML data. Section 6 gives the conclusions.

2 Graph data model

In general, traditional database technologies are always based on a database model. In the case of GDBs, such model uses a kind of a graph.

2.1 Labelled property graphs

Here we will use a (*labelled*) *property graph model* whose basic constructs include:

- entities (nodes),
- properties (attributes),
- labels (types),
- relationships (edges) having a direction, start node, and end node,
- identifiers.

Entities and relationships can have any number of properties, nodes and edges can be tagged with labels. More edges connecting two nodes are allowed. Both nodes and edges are

² <http://neo4j.com/developer/cypher-query-language/> (retrieved on 10.9.2017).

³ <http://tinkerpop.apache.org/> (retrieved on 10.9.2017).

⁴ <http://titan.thinkaurelius.com/> (retrieved on 10.9.2017).

⁵ <https://www.w3.org/TR/2008/REC-xml-20081126/> (retrieved on 10.9.2017).

Fig. 1 Example of a GDB

defined by a unique identifier. Properties are expressed in the key:value style. In graph-theoretic notions we also talk about *labelled and directed attributed multigraphs* in this case. These graphs are used both for GDB and its database schema (if any). An example of a GDB is in Fig. 1.

2.2 XML graphs

In a little restricted version, also XML documents can be represented by labelled, finite trees. More generally, XML documents can have a more general graph structure, due to ID references, but we will not consider this case. Nodes of these data structures are objects, labels on edges are XML tags (element names or attribute names). We will not even consider XML attributes here. Leaves contain atomic values—texts. Figure 2 presents an example of XML document based on the GDB sample in Fig. 1.

XML data can be represented as trees in more ways. One possibility is used in Fig. 3 for XML document in Fig. 2. Comparing to property graphs, these XML trees use only nodes of two types and labelled edges.

XML data can contain more occurrences of one element. To include them into a database, we introduce a set $\mathbf{E}$ of *abstract elements*. An empty abstract element ϵ is also in $\mathbf{E}$. Any function is undefined on ϵ . The set $\mathbf{E}$ serves as a reservoir for construction of XML elements and to their unique identification. Abstract elements can be implemented as inner OIDs in an XML database. The content of an abstract element will be either a string from PCDATA, in the easiest example, or a sequence of abstract subelements, or empty. Neglecting ordering of subelements, the sequence can be replaced by a set. For example, `< town_name > Berlin </town_name >` is an instance element of the `town_name` element object. The `town_name` element object is a (partial) function from $\mathbf{E}$ to PCDATA. A more complex `town` element type can be conceived as a set of functions from $\mathbf{E}$ to the Cartesian product of $\mathbf{E} \times \mathbf{E}$ (abstract town names and

```
<?xml version="1.0" encoding="UTF-8"?>
<teachers>
  <teacher>
    <T_ID>Z121</T_ID>
    <T_name>Uli</T_name>
    <birth_year>1982</birth_year>
    <teaches>
      <day>Thu</day>
      <hour>4</hour>
      <room>S2</room>
      <language>
        <name>English</name>
      </language>
    </teaches>
    <teaches>
      <day>Thu</day>
      <hour>6</hour>
      <room>S1</room>
      <language>
        <name>German</name>
        <textbook>German for beginners</textbook>
      </language>
    </teaches>
    <lives_in>
      <from>30.1.1978</from>
      <town>
        <town_name>Berlin</town_name>
        <population>50</population>
      </town>
    </lives_in>
    <is_born_in>
      <date>23.4.1950</date>
      <town>
        <town_name>Berlin</town_name>
        <population>50</population>
      </town>
    </is_born_in>
  </teacher>
</teachers>
```

Fig. 2 XML document

Fig. 3 Tree representation of XML document in Fig. 2 (a part)

populations). The second component of a `town` couple can be ϵ . Due to the tree representation, for example the town Berlin will be represented twice in such XML tree. Similarly, if more teachers learn German, representation of German language would be repeated in the XML document. This observation reflects the fact that XML data is textual, even for numeric data. Moreover, no conceptual view is supposed for this data.

In [8] we considered only partially ordered XML documents. In paper [9], we proposed an ordered model of XML data, i.e., a modification of its typing apparatus, which preserves ordering of subelements.

3 Modelling and querying graph databases

Current commercial GDBMSs need more improvements to meet traditional definitions of conceptual and database schema known, e.g., from the relational databases world. The graph database model is usually not presented explicitly, but it is hidden in constructs of a data definition language (DDL) which is at disposal in the given GDBMS. These languages also enable to specify some simple ICs. Conceptual modelling of graph databases is not used at all. An exception is the GRAD database model [6], which although schema-less, uses conceptual constructs occurring in E-R conceptual model and some powerful ICs. Both graph conceptual schema and graph database schema can provide effective communication medium between users of any GDB. They can also significantly help to GDB designers.

In Sect. 3.1 we propose a graph data modelling based on property graphs and introduce a conceptual level for this purpose. Section 3.2 repeats some basics of XML modelling with the help of DTD⁶ language. To complete graph modelling issues, we mention ICs in GDB in Sect. 3.3. Section 3.4 introduces some principles of graph querying.

3.1 Modelling graph data based on property graphs

In [12] we proposed a binary E-R model as a variant for graph conceptual modelling considering strong entity types, weak entity types, relationship types, attributes, identification keys, partial identification keys, ISA-hierarchies, and min-max ICs. The notation is based on the Oracle Designer CASE [3]. The graph conceptual schema in Fig. 4 uses for min-max ICs well-known notation with dotted lines and crow's foot's used for the start node and the end node of some edges. The perpendicular line denotes the identification and existence dependency of weak entity types. Subtyping (ISA-hierarchies) is simply expressed by arrow to the entity supertype. Relationship types are expressed by couples of mutually “inverse” labels, e.g., (*Is_thought_by*, *Teaches*).

Figures 4 and 5 give examples of graph conceptual schema and graph database schema (without attributes/properties), respectively. Graphical min-max ICs (see, Fig. 2) can be expressed equivalently by expressions ($E_1:(a, b)$, $E_2:(c, d)$), where $a, c \in \{0, 1\}$, $b, d \in \{1, n\}$, and n means “any number greater than 1”. For example, ($Teacher:(1, 1)$,

Fig. 4 Graph conceptual schema

Fig. 5 Graph database schema without properties

$Town:(0, n)$) can be associated with the relationship type (*Is_born_in*, *Is_birthplace_of*). Expressions ($E_1:(a, n)$, $E_2:(c, n)$) correspond to cardinalities $m : n$ in an alternative notation.

A correct graph conceptual schema may be mapped into an equivalent (or nearly equivalent) graph database schema with the straightforward mapping algorithm [12] but with a weaker notion of a database schema, i.e., some inherent ICs from the conceptual level have to be neglected to satisfy usual notation of labelled property graphs. Consequently, we can propose several different graph database schemas from a graph conceptual schema. For example, the edges *Teaches* and *Is_born_in* provide only a partial information w.r.t. the associated source conceptual schema. The inverted arrow *Is_taught_by* could be used as well. Due to the loss of the inherent ICs occurring at the conceptual level, we should put some explicit ICs into the GDB schema, e.g., that “A teacher can teach more languages” and “A teacher is born in exactly one town”.

As usual, only single-valued attributes of form *key: domain* are considered here, where domains include elementary descriptive types like *String*, *Number*, *Date*, etc. Then, the identification key of *Teacher*

⁶ <https://www.w3.org/TR/xml11/> (retrieved on 10.9.2017).

Fig. 6 Graph database schema with properties

```
<!DOCTYPE teachers [
<!ELEMENT teachers(teacher*)
<!ELEMENT teacher (T_ID,T_name,birth_year,teaches*,lives_in, is_born_in)
<!ELEMENT teaches (day,hour,room,language)>
<!ELEMENT language(name,textbook?)
<!ELEMENT lives_in(from,town)>
<!ELEMENT is_born_in(date,town)>
<!ELEMENT town(town_name,population?)>
<!ELEMENT town_name (#PCDATA) >
<!ELEMENT population (#PCDATA) >
...
<!ELEMENT birth_year (#PCDATA) >
]>
```

Fig. 7 DTD for the XML document in Fig. 2

could be #Person_ID. On the database schema level, the identification key of Street would be {Town_name, Street_name}. Details of mapping of graph conceptual schemas to graph database schemas can be found in [12].

Figure 6 presents the graph database schema associated to the GDB in Fig. 1. Obviously, the schema is again a labelled and directed attributed multigraph.

The values of some properties can be unknown or undefined in a GDB. This reminds NULL values in SQL databases. For example, Textbook:NULL could be considered. In GDB as well as in NoSQL databases generally, such properties are not explicitly represented. In Fig. 1, the Language node for English is the case.

3.2 Modelling graph data based on XML format

The simplest way how to model XML data on the database schema level is given by the language DTD. A more rich collection of modelling tools is offered by XML Schema language⁷. The DTD subset considered in our examples uses only element declarations. Figure 7 contains a DTD describing the XML data in Fig. 2.

3.3 Integrity constraints in graph databases

Due to the graph structure of data in GDB, associated explicit ICs can have also a graph form. Very simple IC is a *functional dependency* (FD) or *conditional functional*

dependency (CFD) introduced in [12]. For example, “Each teacher is born in one town” and “Teachers born later than in 1994 teach at most one language” are examples of FD and CFD, respectively. A usable approach is also offered by above mentioned GRAD database model. It enables to express some semantic restrictions over the graph data with using graph patterns. A graph pattern P is a predicate on the graph topology (specifying conditions on the structural properties of the graph) and properties (specifying conditions on their values) of the graph elements.

At least inherent ICs coming from a graph conceptual schema should be considered as explicit ICs on the graph databases level, i.e., using a DDL for their formulation. In [13], we have focused on graph database Neo4j and its possibilities to express a database schema and/or ICs. We have extended these possibilities through new constructs in Neo4j DDL including their prototype implementation and experiments.

On the other hand, NoSQL databases often does not require the notion of a graph database schema at all. Strict application of schemas is sometimes considered disadvantageous by those who develop applications for dynamic domains, e.g., domains working with user-generated content, where the data structures are changing very often [1]. Consequently, many GDBMSs are schema-less, including Neo4j. OrientDB⁸ even distinguishes three roles of graph database schema: schema-full, schema-less, and schema-hybrid. Even schema-less, some GDBMSs support to specify some types of ICs. Neo4j uses for this purpose the language Cypher.

3.4 Graph querying

In this section, we focus on basics of graph querying (for more details see [11]). Its simplest type uses the index-free adjacency. In practice, the basic queries like *k-hop queries* are the most frequent. Looking for a node, looking for its neighbours (1-hop query), scan edges in several hops, retrieval of property values, etc., belong into the category.

More complex queries are *subgraph* and *supergraph queries*. They belong to traditional queries based on exact matching. Other typical queries include *breadth-first/depth-first search*, *path* and *shortest path finding*, *least-cost path finding*, *finding cliques* or *dense subgraphs*, *finding strong connected components*, etc.

Very useful are regular path queries (RPQ). RPQs have the form:

$$\text{RPQ}(x, y) := (x, R, y)$$

⁷ <https://www.w3.org/XML/Schema> (retrieved on 10.9.2017).

⁸ <http://www.orientdb.com/> (retrieved on 10.9.2017).

where R is a regular expression over the vocabulary S of edge labels. Construction of regular expressions is as follows:

$$R::=s|R.R|R|R^*|R?(R)$$

where s is a label from S . RPQs provide couples of nodes connected by a path conforming to R . With the closure of RPQs under conjunction and existential quantification we obtain *conjunctive* RPQs.

For example, the Cypher working with Neo4j databases lacks some fundamental graph querying functionalities, namely, RPQs and graph construction. In [2] an interesting newer approach is offered by the language G-Path. G-Path is an RPQ language working on graphs, which supports mostly all useful regular expression operators. PGQL [19] is based on the paradigm of graph pattern matching, closely follows syntactic structures of SQL, and provides RPQs with conditions on labels and properties.

4 Database modelling with functions

Our approach to graph modelling and querying is based rather on conceptual structures than database ones. We start with a conceptual modelling can based on the notion of attribute viewed as an empirical typed function that is described by an expression of a natural language [7]. A lot of papers are devoted to this approach developed and studied mainly in the 1990s (see, e.g., [10]). The type apparatus allows a number variants. For our purposes, we can handle with functional types and tuple types built in a hierarchical way.

4.1 Types

A hierarchy of types is constructed as follows. We assume the existence of some (*elementary*) types $S_1, \dots, S_k$ ($k \geq 1$). They constitute a base $\mathbf{B}$. More complex types are constructed in the following way.

If $S, R_1, \dots, R_n$ ($n \geq 1$) are types, then

- (i) $(S : R_1, \dots, R_n)$ is a (*functional*) type,
- (ii) $(R_1, \dots, R_n)$ is a (*tuple*) type.

The set of *types* $\mathbf{T}$ over $\mathbf{B}$ is the least set containing types from $\mathbf{B}$ and those given by (i)-(ii). When S_i in $\mathbf{B}$ are interpreted as non-empty sets, then $(S : R_1, \dots, R_n)$ denotes the set of all (total or partial) functions from $R_1 \times \dots \times R_n$ into S , $(R_1, \dots, R_n)$ denotes the Cartesian product $R_1 \times \dots \times R_n$.

In the conceptual modelling, each base $\mathbf{B}$ consists of descriptive and entity types. Descriptive types (*String*, *Number*, etc.) serve for domains of properties. Their elements are constants like ‘*Prague*’, ‘*John*’, ‘*201400*’, etc. The elementary type $\text{Bool} = \{\text{TRUE}, \text{FALSE}\}$ is also in $\mathbf{B}$. The

type Bool allows to type some objects as sets and relations. They are modelled as unary and n -ary characteristic functions, respectively. The concept of the set then becomes redundant.

The fact that X is an object of type $R \in \mathbf{T}$ can be written as XR , or “ X is the R -object”. For each typed object o the function *type* returns $\text{type}(o) \in \mathbf{T}$ of o . Logical connectives, quantifiers and predicates are also typed functions: e.g., **and**/($\text{Bool} : \text{Bool}, \text{Bool}$), R -identity $=_R$ is $(\text{Bool} : R, R)$ -object, universal R -quantifier Π_R , and existential R -quantifiers Σ_R are $(\text{Bool} : (\text{Bool} : R))$ -objects. R -singularizer $I_R/(R : (\text{Bool} : R))$ denotes the function whose value is the only member of an R -singleton and in all other cases the application of I_R is undefined. Arithmetic operations $+$, $-$, $*$, $/$ are examples of $(\text{Number} : \text{Number}, \text{Number})$ -objects. We can also type functions of functions etc.

4.2 Attributes

Object structures usable in building a database can be described by some expressions of a natural language. For example, “the language thought by a teacher at a school” (abbr. LTS) is a $(\text{Language} : \text{Teacher}, \text{School})$ -object, i.e., a (partial) function $f : \text{Teacher} \times \text{School} \rightarrow \text{Language}$, where *Teacher*, *School*, and *Language* are appropriate elementary types. Such functions are called *attributes* in [7].

More formally, attributes are functions of type $((S : T) : W)$, where W is the logical space (possible worlds), T contains time moments, and $S \in \mathbf{T}$. M_w denotes the application of the attribute M to $w.W$, M_{wt} denotes the application of M_w to the time moment t . In our approach to conceptual modelling, we can omit parameters w and t in $\text{type}(M)$. In the case of LTS attribute, we suppose only possible worlds, where teachers teach at most on language in each school. However, this may not be true for other possible worlds. For GDBs we can elementary entity types conceive as sets of node IDs. For a better readability, *John*, *Prague*, *Frank* can be simply such IDs.

Attributes can be constructed according to their type in a more complicated way. For example, “the classes in a school” could be considered as an attribute CS of type $((\text{Bool} : (\text{Bool} : \text{Student})) : \text{School})$, i.e., the classes contain sets of students and the CS returns a set of classes (of students) for a given school. Obviously, *Student* is an elementary type.

We can also consider other functions that need no possible world. For example, aggregate functions like $\text{COUNT}_S/((\text{Number} : (\text{Bool} : S)), \text{SUM}/(\text{Number} : (\text{Bool} : \text{Number}))$ ⁹ and arithmetic operations provide such func-

⁹ We suppose here, that SUM is defined on sets only.

tions. These functions have the same behaviour in all possible worlds and time moments. Consequently, we can distinguish between two categories of functions: *empirical* (e.g. attributes) and *analytical*. The former are conceived as partial functions from the logical space. Range of these functions are again functions. Analytical functions are of type S , where S does not depend on W and T .

The notion of attribute applied in GDBs could be restricted on attributes of types $(R:S)$, $((Bool:R):S)$, or $(Bool:R,S)$, where R and S are entity types. This strategy simply covers binary functional types, binary multivalued functional types, and binary relationships described as binary characteristic functions. The last option corresponds to $m:n$ relationship types. For modelling directed graphs the first two types are sufficient, because $m:n$ relationships types can be expressed by two „inverse“ binary multivalued functional types. Here we will consider always one of them (see, e.g., Teaches below). Thus, a graph database schema can reflect a reality only partially, similarly to non-functional graph data modelling.

Now we add properties. Properties describing entity types can be of types $(S_1, \dots, S_m:R)$, where S_i are descriptive elementary types and R is an entity type. So we deal with functional properties. Similarly, we can express properties of edges. They are of types $(S_1, \dots, S_m, R_1:R_2)$ and $((Bool:S_1, \dots, S_m, R_1):R_2)$ for binary functional and binary multivalued functional types, respectively.

Then a functional database schema describing GDB in Fig. 1 can look as:

```
La/((Name, Textbook):Language)
Te/((T_ID, T_Name, Birth_year):Teacher)
Tw/((Town_name, Population):Town)
Teaches/((Bool:Day, Hour, Room,
Language):Teacher)
Is_Born_in/((Date, Town):Teacher)
Lives_in/((From, Town):Teacher)
```

where *La*, *Te*, *Tw*, *Teaches*, *Is_Born_in* and *Lives_in* are typed variables. The use of variables correspond to a “database view” of the modelled world. Since we do not consider temporal functional databases here, it is not necessary to use time explicitly.

We remark, however, that our functional GDBs with such schemas can contain isolated nodes with at least one property. IDs of edges are not necessary, because edges are not explicitly considered.

Thus, a *functional graph database schema* is a set of variables of types from T restricted to attribute types. A *functional graph database* is any evaluation of these variables. Thus, certain variables serve for referencing the associated database. For convenience, we denote the variables from the functional graph database schema by names remaining entity

and relationship types in the associated traditional GDB schema or at least their abbreviations.

5 Manipulating functions

A manipulating language for functions is traditionally a typed lambda calculus. Our version of the typed lambda calculus directly supports manipulating objects typed by T introduced in Sect. 4.1. We will suppose a collection **Func** of constants, each having a fixed type, and denumerable many variables of each type. Then the *language of (lambda) terms* LT is defined as follows:

Let types $R, S, R_1, \dots, R_n$ ($n \geq 1$) are elements of T .

- (1) Every variable of type R is a term of type R .
- (2) Every constant (a member of **Func**) of type R is a term of type R .
- (3) If M is a term of type $(S:R_1, \dots, R_n)$, and $N_1, \dots, N_n$ are terms of types $R_1, \dots, R_n$, respectively, then $M(N_1, \dots, N_n)$ is a term of type S . */application/*
- (4) If $x_1, \dots, x_n$ are different variables of the respective types $R_1, \dots, R_n$ and M is a term of type S , then $\lambda x_1, \dots, x_n(M)$ is a term of type $(S:R_1, \dots, R_n)$. */lambda abstraction/*
- (5) If $N_1, \dots, N_n$ are terms of types $R_1, \dots, R_n$, respectively, then $(N_1, \dots, N_n)$ is a term of type $(R_1, \dots, R_n)$. */tuple/*
- (6) If M is a term of type $(R_1, \dots, R_n)$, then $M[1], \dots, M[n]$ are terms of respective types $R_1, \dots, R_n$. */components/*

5.1 From the LT language to a more user-friendly notation

Instead of the position notation we can use more readable dot notation for components. Consider the Prague object (entity, node). Then instead of $Tw(\text{Prague})$ [2], where $\text{Prague}/\text{Town}$, we can write $Tw(\text{Prague}).\text{Population}$. The real effect of this convention is that we approach *Population* property independently on its position in the tuple described by the associated functional database schema. In the case that the population is not present between properties of the town Prague, the function will be undefined.

Terms are interpreted in a standard way by means of an interpretation assigning to each function symbol from **Func** an object of the same type, and by a semantic mapping $[]$ from LT into all functions given by T . **Func** influences the expressive power of LT. It can contain usual arithmetic and aggregation functions, etc. Of a special importance is R -identity $=_{(Bool:(Bool:S),(Bool:S))}$ usable for a comparison of two sets of S -objects. We will denote the $(Bool:(Bool:S), (Bool:S))$ type as $2sets$ for

better readability. Consider Teacher-objects John and Frank. Then with

$$\lambda l_1^{\text{Language}} \exists d_1, h_1, r_1, \text{Teaches}(\text{John})(d_1, h_1, r_1, l_1^{\text{Language}}) =_{2\text{sets}} \lambda l_2^{\text{Language}} \exists d_2, h_2, r_2, \text{Teaches}(\text{Frank})(d_2, h_2, r_2, l_2^{\text{Language}})$$

we can test whether John and Frank teach the same set of languages. Similarly to domain relational calculus, we can simplify this expression by omitting some existential quantifiers and associated variables. Then the resulted expression can look as

$$\lambda l_1^{\text{Language}} \text{Teaches}(\text{John})(l_1^{\text{Language}}) =_{2\text{sets}} \lambda l_2^{\text{Language}} \text{Teaches}(\text{Frank})(l_2^{\text{Language}})$$

A further simplification could be made by omitting lambda abstraction supposing that information about objects considered is in the $\text{Bool}(\text{Language})$ -identity. Then we can write

$$\text{Teaches}(\text{John}) =_{2\text{sets}} \text{Teaches}(\text{Frank})$$

In other words, an application is evaluated as the application of an associated function to given arguments, a lambda abstraction “constructs” a new function. In the conventional approach a valuation δ is used. Supposing this mapping we can assign objects to every variable occurring in a term.

For example, $\text{CS}(\text{Oxford_House})('AM_training')$ is *TRUE*, if there is the class *AM_training* in the *Oxford_house* school (*'AM_training'* is a constant of type $(\text{Bool} : \text{Student})$). It is true while *'AM_training'* contains all students of the class *AM_training*.

In accordance with the semantics of the quantifiers and the singularizer, we can write simply $\forall x(M)$ instead of $\Pi(\lambda x(M))$. Similarly, $\exists x(M)$ replaces $\Sigma(\lambda x(M))$. Finally, we write $I(\lambda x(M))$ shortly as *onlyone* $x(M)$ and read “the only x such as M ”. Certainly, M/Bool .

From the database point of view, we have at disposal a powerful declarative language for formulating schema transformation, ICs, etc., even querying GDBs, as we shall see in Sect. 5.1. Section 5.2 shows that a similar technique can be used in context of XML trees.

5.2 Querying graph data functionally

The LT language can be used as a theoretical tool for building a functional database language. A query in such language is expressed by a term of LT, e.g.,

$$\lambda t, n(n = \text{COUNT}_{\text{Language}}(\lambda l(\exists d, h, r \text{Teaches}(t)(d, h, r, l))))$$

with d, h, r , and l of types *Teacher*, *Day*, *Hour*, *Room*, *Language*, respectively.

Using the syntactic abbreviation similar to that one used in Section 4.3, the resulted lambda expression

$$\lambda t^{\text{Teacher}}, n^{\text{Number}}(n^{\text{Number}} = \text{COUNT}_{\text{Language}}(\lambda l(\text{Teaches}(t^{\text{Teacher}})(l^{\text{Language}}))))$$

denotes the query “Give a set of couples associating with each teacher the number of languages teaching by him/her”. Obviously, the query could be also reformulated as $\lambda t(\lambda n(\dots))$, or more conventionally $\lambda t \text{ onlyone } n(\dots)$.

To consider only teachers born in Prague the lambda expression might look like

$$\lambda t^{\text{Teacher}}, n^{\text{Number}}(n^{\text{Number}} = \text{COUNT}_{\text{Language}}(\lambda l(\text{Teaches}(t^{\text{Teacher}})(l^{\text{Language}})) \text{ and } l = \text{born_in}(t^{\text{Teacher}}).\text{Town_name} = 'Prague'))$$

Remark 1 In our conceptual framework we could conceive each query as an attribute, i.e., a lambda expression dependent on possible worlds and time moments. In practice, we omit $\lambda w(\lambda t \dots)$ in its head. Clearly, the resulted lambda expressions can express more complex graph structures than k -hop queries, i.e., they can contribute to constructions of new graphs from the original GDB.

In querying GDBs, RPQs are of user importance. We will consider expressions

$$\lambda x, y \text{Reg}(x, y)$$

where *Reg* is an expression simulating concatenation and closure. There are two styles how to construct *Reg*. First, consider the attribute

$$\text{Friends_Of_Friend}/((\text{Bool} : \text{Person}) : \text{Person}),$$

abbreviated as *FOF*. $\text{FOF}(p_i)(p_j) = \text{TRUE}$ expresses the fact that there is an edge between p_i and p_j in the associated GDB. Then

$$\text{FOF}^*(p_1, p_k)$$

will denote the expression $\text{FOF}(p_1)(p_2) \text{ and } \dots \text{ and } \text{FOF}(p_{k-1})(p_k)$, for a k ($k > 1$), and

$$\lambda x, y \text{FOF}^*(x, y)$$

provides a set of couples (p_1, p_k) , where there is a directed path from p_1 to p_k along edges of *FOF*.

Now we will consider a single-valued function, e.g.,

$$\text{Manager_of}/(\text{Person} : \text{Person}),$$

abbreviated as MO . $MO^*(p)$, where p/Person , will denote the expression $MO(\dots(MO(p)\dots))$ with l applications of MO , for a $l \geq 1$, and

$$\lambda x, y (MO^*(x) = y)$$

provides a set of couples (p_1, p_k) , where there is a directed path from p_1 to p_k along edges MO .

A composition of different functions representing relationships and properties looks, e.g., as follows. The following term expresses a YES/NO query “Is John born in Berlin?”

$$Tw(Is_born(\text{John}).\text{Town}).\text{Town_name} = 'Berlin')$$

One of the most fundamental problems in graph processing is pattern matching. Specifically, a pattern match query searches over a GDB to look for the existence of a pattern graph in a GDB. For example, triangle counting is used heavily in social network analysis. It is easy to formulate terms providing sets of triples (p_1, p_2, p_3) and there is a path $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_1$ in GDB. Clearly, we have to suppose oriented labelled edges, i.e., attribute names, e.g., FOF . Finding only structural triangles, regardless of edge names, would require to expand the typing of variables and to move to the second-order typed lambda calculus.

We remind that the LT is not computationally complete. But it makes possible to increase its computational power by adding new built-in functions into **Func**. In other words, LT is extendible with various mathematical functions, including logical operators.

5.3 Querying XML trees functionally

First, we extend the type system defined in Sect. 4.1 by the union type $(T_1 + T_2)$ denoting union of sets T_1 and T_2 . Now we introduce the type system $\mathbf{T}_{\text{reg}}$. Let $\mathbf{B} = \{\text{PCDATA}, \text{BOOL}, \text{NAME}\}$. The type system $\mathbf{T}_{\text{reg}}$ over $\mathbf{B}$ is recursively defined as follows.

$$\begin{aligned} T ::= & \text{tag: PCDATA} \mid \text{tag: elementary regular expression} \\ & \text{where tag} \in \text{NAME.} \\ & \mid T^* \quad \text{zero or more (Kleene closure)} \\ & \mid T^+ \quad \text{one or more (positive closure)} \\ & \mid T? \quad \text{zero or one} \\ & \text{where } T \text{ is an alternative or elementary regular expression.} \\ & \mid (T_1 \mid T_2) \quad \text{alternative} \end{aligned}$$

The type system $\mathbf{T}_{\text{reg}}$ describes *regular expressions* over character data, similarly as it is allowed in DTDs. Suppose that PCDATA is interpreted as a set of character data (strings). Then tag: PCDATA denotes the set of character data labelled by tag . The type tag: denotes the empty labelled character object. $(T_1 \cup T_2)$ denotes the set of objects of type $T_1 \cup T_2$. T^*

is an abbreviation for $(\text{BOOL} : T)$. In this case, $(\text{BOOL} : T)$ -objects include $\emptyset$ which simulates the empty set. Similarly, T^+ denotes the set of $(\text{BOOL} : T)$ -objects except $\emptyset$, and $T?$ the set of objects of type $T \cup \text{NIL}$.

Now we will define (XML) element types, following the established type system $\mathbf{T}_{\text{reg}}$ over $\mathbf{B}$. In order to distinguish tags used in $\mathbf{T}_{\text{reg}}$ from element tags we will suppose for each tag used in the elementary types $\text{tag:}T$ or tag: the existence of the TAG name denoting an associate element type¹⁰. The same holds for any $\text{tag} \in \text{NAME}$. Thus, NAME contains both tags and TAGs.

Let $\mathbf{T}_{\text{reg}}$ over $\mathbf{B}$ be the type system and $\mathbf{E}$ be a set of abstract elements. Then the type system $\mathbf{T}_{\mathbf{E}}$ induced by $\mathbf{T}_{\text{reg}}$ (or $\mathbf{T}_{\mathbf{E}}$ if $\mathbf{T}_{\text{reg}}$ is understood) containing the *regular element expressions* is given by the following grammar:

$$\begin{aligned} E ::= & \text{TAG:}T \mid \text{TAG:} \quad \text{elementary element types} \\ & \text{where tag:}T \text{ and tag: are elementary regular expressions over } \mathbf{B} \\ & \mid E^* \mid E^+ \mid E? \mid (E_1 \mid E_2) \\ & \mid \text{TAG:}(E_1, \dots, E_n) \\ & \text{where tag} \in \text{NAME.} \end{aligned}$$

Elementary element types and regular element expressions $\text{TAG:}(E_1, \dots, E_n)$ are called *element types*.

The functional semantics of the element types associate with TAG:PCDATA the set of all functions from $\mathbf{E}$ to tag: PCDATA . For a non-elementary element type T , the semantics of $\text{TAG:}T$ are also functional, but the functions are more complex.

Then an *XML-database schema*, $\mathbf{S}_{\text{XML}}$, is a set of variables of types from $\mathbf{T}_{\mathbf{E}}$. Given an *XML-database schema* $\mathbf{S}_{\text{XML}}$, an *XML-database* is any valuation of these variables. Thus, certain variables serve for referencing the associated database. For convenience, we denote the variables from $\mathbf{S}_{\text{XML}}$ by the same names as TAGs from $\mathbf{T}_{\mathbf{E}}$, e.g. BOOK, AUTHOR, etc. For example, a number of types associated with DTD in Fig. 7 can look as follows:

```
TEACHERS : (TEACHER*)
TOWN : (TOWN_NAME, POPULATION?)
TOWN_NAME : PCDATA
```

To manipulate XML data from XML-database based on $\mathbf{T}_{\mathbf{E}}$, it is necessary only to extend the original LT language by tagged terms $K:M$, where K/NAME . If M/T , then $K:M/(T:E)$. The resulted language version called XML- λ in [8] enables to simulate much of the XPath¹¹ language as well as the 1st order logic, aggregation functions, arithmetic, and user defined functions. Obviously, a more user-friendly notation

¹⁰ This distinguishing is only formal and can be done in any other way in practice.

¹¹ <https://www.w3.org/TR/xpath/> (retrieved on 10.9.2017).

can be used. For example,

```
POPULATION(TOWN(IS_BORN_IN(TEACHER(e))))
```

can be rewritten as

```
e.TEACHER.IS_BORN_IN.TOWN.POPULATION
```

When we use a path in more logical conditions, it is possible to write the common prefix only once in XML- λ and to put conditions in parentheses. For example,

```
 $\lambda x$ (TEACHER(IS_BORN_IN.TOWN.TOWN_NAME =  
LIVES_IN.TOWN.TOWN_NAME and T_NAME = x))
```

denotes the query “Give a set of teachers names who live in the same town where they were born”.

6 Conclusions

The objective of this paper was to provide an alternative approach to GDB querying based on a functional approach. Comparing to other graph query languages, the functional language LT designed here is based on the notion of graph conceptual schema using the notion of attribute. The approach is based on the idea that the conceptual view can be directly conceived as a database view. More technically, only an appropriate database implementation of conceptual structures is necessary. Query languages based on this approach are usable in environments where GDB is searched for collecting and aggregating information from nodes and relationships rather than extractions of only structural patterns.

We have seen that not only labelled property graphs but also XML trees provide application environment for the functional approach. Only the type system is different.

Finally, a few words about usability of functional querying in GDBs. All the techniques associated with GDBMS and supported in any graph search engine should fulfil so called FAE rule [5]. The FAE rule says that the quality of search engines includes three key factors: *Friendliness*, *Accuracy* and *Efficiency*, i.e., that a good search engine must provide the users with a friendly query interface and highly accurate answers in a fast way. A friendliness of our functional language is still missing till now. This is the main challenge for future work.

Acknowledgements This work was supported by the Charles University project Q48.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution,

and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

- Angels, R.: A comparison of current graph database models. In: IEEE 28th Int. Conference on Data Engineering Workshops, pp. 171–177 (2012)
- Bai, Y., Wang, Ch., Ning, Y., Wu, H., Wang, H.: G-path: flexible path pattern query on large graphs, pp. 333–336. WWW (Companion Volume) (2013)
- Barker, R.: Case*Method: Entity Relationship Modeling. Addison-Wesley Publ. Comp., Boston (1990)
- Gray, P.M.D., Kerschberg, L., King, P.J.H., Pouloussilis, A.: The Functional Approach to Data Management. Springer, Berlin (2004)
- Ma, S., Li, J., Hu, Ch., Lin, X., Huai, J.: Big graph search: challenges and techniques. Front. Comput. Sci. **10**(3), 387–398 (2016)
- Ghrab, A., Romero, O., Skhiri, S., Vaisman, A., Zimányi, E.: GRAD: On Graph Database Modeling. Cornell University Library (2016). [arXiv:1602.00503](https://arxiv.org/abs/1602.00503)
- Pokorný, J.: A function: unifying mechanism for entity-oriented database models. In: Batini, C. (ed.) Entity-Relationship Approach, pp. 165–181. Elsevier Science Publishers B.V, North-Holland (1989)
- Pokorný, J.: XML functionally. In: Desai, B.C., Kiyoki, Y., Toyama, M. (eds.) Proc. of IDEAS2000, pp. 266–274. IEEE Comp. Society (2000)
- Pokorný, J.: XML querying with functions. In: Kiyoki, Y. et al (eds.) Proc. of the IASTED Int. Conf. Information Systems and Databases, pp. 204–209. Acta Press (2002)
- Pokorný, J.: Database semantics in heterogeneous environment. In: Jeffery, K.G., Král, J., Bartošek M. (eds.) Proc. of 23rd Seminar SOFSEM96: Theory and Practice of Informatics, pp. 125–142. Springer-Verlag (1996)
- Pokorný, J.: Graph databases: their power and limitations. In: Saeed, K. and Homenda, W. (eds.) Proc. of 14th Int. Conf. on Computer Information Systems and Industrial Management Applications (CISIM), LNCS 9339, pp. 58–69. Springer, Berlin (2015)
- Pokorný, J.: Conceptual and database modelling of graph databases. In: Desai, B. (ed.) Proc. of IDEAS’ 16, pp. 370–377. ACM (2016)
- Pokorný, J., Valenta, M., Kovačič, J.: Integrity constraints in graph databases. In: Proc. of the 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017), 7th Int. Symp. on Frontiers in Ambient and Mobile Systems (FAMS 2017), pp. 975–981. Elsevier Science, Procedia Computer Science (2017). <https://doi.org/10.1016/j.procs.2017.05.456>
- Pokorný, J.: Functional Querying in Graph Databases. In: Nguyen N., Tojo S., Nguyen L., Trawiński B. (eds.) Proc. of 9th Asian Conference on Intelligent Information and Database Systems (ACIIDS 2017), pp. 291–301, Part I, LNCS 10191. Springer (2017)
- Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O’Reilly Media (2013)
- Shipman, D.W.: The functional data model and the data languages DAPLEX. ACM Trans. Database Syst. (TODS) **6**(1), 140–173 (1981)
- Stanek, G., Kolmar, S.: How Neo4j co-exists with Oracle RDBMS. White paper, Neo4j (2016)
- Tivari, S.: Professional NoSQL. Wiley/Wrox (2011)

19. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In: Proc. of the 4th Int. Workshop on GRADES, Redwood Shores, CA (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.